UNIVERSIDAD POLITECNICA DE MADRID

Facultad de Informática

TESIS DOCTORAL

Estructuras óptimas de detección y corrección de errores dirigidas a computadores construídos con lógica de alto nivel de integración. Aplicación industrial.

Antonio Pérez Ambite

Madrid, Febrero de 1982

TESIS DOCTORAL

Estructuras óptimas de detección y corrección de errores dirigidas a computadores construídos con lógica de alto nivel de integración. Aplicación industrial.

presentada en la
FACULTAD DE INFORMATICA DE MADRID

para la obtención del grado de
DOCTOR EN INFORMATICA

Autor: ANTONIO PEREZ AMBITE
Ingeniero de Telecomunicación

Director: D. PEDRO DE MIGUEL ANASAGASTI
Catadrático de Computadores de la
Facultad de Informática de Madrid

Madrid, Febrero de 1982

I.S.B.N.: 84-85632-29-X
Depósito Legal.: M-8579-1982

Imprime,Edita y Distribuye:Dpto.Publicaciones de la Facultad de Informática
bajo el patrocinio de la Fundación General de la Universidad Politécnica de
Madrid.

TESIS DOCTORAL

Estructuras óptimas de detección y corrección de errores dirigidas a computadores construidos con lógica de alto nivel de integración. Aplicación industrial.

TRIBUNAL CALIFICADOR

Presidente: D. RAFAEL PORTAENCASA BAEZA

Vocales: D. ANTONIO INSUA NEGRAO

D. EUGENIO ANDRES PUENTE

D. PEDRO DE MIGUEL ANASAGASTI

D. RAMON PUIGJANER TREPAT

Con el presente trabajo, presentado el dia 9 de Marzo de 1982 en la Facultad de Informática de Madrid, obtuvo, el autor, el Grado de Doctor en Informática con la calificación de Sobresaliente Cum Laude.

Este trabajo ha podido ser realizado gracias al interés y aliento prestados por el director del mismo D. Pedro de Miguel Anasagasti. Es también de destacar la colaboración de D. Javier Montes Alonso y D. Manuel Guerra Tejado en la aplicación práctica. Asimismo debo remarcar el apoyo recibido de mis compañeros de trabajo, y en particular el de aquellos que desde el Centro de Documentación de esta Facultad han proporcionado la base bibliográfica de la presente Tesis.

A todos ellos deseo expresar mi mas sincero agradecimiento.

Madrid, 18 de Febrero de 1982.

RESUMEN

Este trabajo trata de la aplicación de los códigos detectores y correctores de error al diseño de los Computadores Tolerantes a Fallos, planteando varias estrategias óptimas de detección y corrección para algunos subsistemas.

En primer lugar, se justifica la necesidad de aplicar técnicas de Tolerancia a Fallos. A continuación se hacen previsiones de evolución de la tecnología de integración, así como una tipificación de los fallos en circuitos integrados. Partiendo de una recopilación y revisión de la teoría de códigos, se hace un desarrollo teórico cuya aplicación permite obligar a que algunos de estos códigos sean cerrados respecto de las operaciones elementales que se ejecutan en un computador. Se plantean estrategias óptimas de detección y corrección de error para sus subsistemas mas importantes, culminando en el diseño, realización y prueba de una unidad de memoria y una unidad de proceso de datos con amplias posibilidades de detección y corrección de errores.

ABSTRACT

The present work deals with the application of error detecting and correcting codes to the design of Fault Tolerant Computers. Several optimum detection and correction strategies are presented to be applied in some subsystems.

First of all, the necessity of applying Fault Tolerant techniques is explained. Later, a study on integration technology evolution and typification of integrated circuit faults is developed. Based on a compilation and revision of Coding Theory, a theoretical study is carried out. It allows us to force some of these codes to be closed over elementary operations. Optimum detection and correction techniques are presented for the most important subsystems. Finally, the design, building and testing of a memory unit and a processing unit provided with wide error detection and correction posibilities is shown.

INDICE.

CAPITULO I

INTRODUCCION

1.1. Tolerancia a fallos.

1.1.1. Justificación de la Tolerancia a Fallos.

El problema de la fiabilidad de los computadores ha preocupado tanto a diseñadores como a usuarios desde la aparición de éstos, puesto que, como cualquier equipo electrónico, pueden sufrir en un momento dado un fallo en algún componente.

Durante su primera generación los computadores se construían a base de componentes con fiabilidad muy baja (relés, tubos de vacio, etc.), por tanto, el tiempo entre fallos era muy pequeño. Comenzaron a utilizarse entonces técnicas de detección y recuperación de errores para aumentar la fiabilidad.

La aparición del transistor, componente mucho mas fiable, dio lugar a que durante la segunda generación de computadores perdiera énfasis la aplicación de técnicas de detección y recuperación de errores. Unicamente se utilizaban programas de diagnóstico y programas de ayuda a mantenimiento. Esta técnica sigue utilizándose actualmente en forma de "microdiagnóstico".

La solución normalmente aplicada para paliar el funcionamiento erróneo de un sistema, debido a la aparición de un fallo, es proceder a una operación manual de mantenimiento, con la que se repara el mismo, quedando así listo para funcionar correctamente hasta que se produzca un nuevo fallo. En algunos casos, la aplicación de esta reparación manual es insuficiente por varias razones:

-Retrasos e interrupciones inaceptables en procesos de tiempo real, provocados por dicha reparación manual.

-Inaccesibilidad de algunos sistemas.

-Excesivo costo de mantenimiento en algunas instalaciones.

Además, durante la pasada década, las áreas de aplicación de los computadores se han extendido a campos en los que la fiabilidad es un factor crítico. Así pues, existen áreas donde resulta inadmisible incluso la aparición o existencia de un fallo. Algunas de éstas son:

-Aplicaciones en las que un fallo pueda poner en peligro una vida humana. Por ejemplo, control de tráfico aéreo, control de trenes, control de unidades de cuidados intensivos en hospitales, etc.

-Aplicaciones en las que un fallo imprevisto pueda suponer un gran quebranto económico. Por ejemplo, control de sistemas de conmutación de líneas telefónicas, control de procesos en factorías automatizadas, etc.

-Uso de sistemas en lugares inaccesibles para un mantenimiento manual, tales como satélites artificiales, plantas subacuáticas, etc.

Todas estas áreas de aplicación, junto con otras muchas exigen niveles de fiabilidad tan sumamente altos, que resulta imposible alcanzarlos utilizando métodos clásicos. Así pues, es necesario introducir en el diseño de computadores las técnicas de "Tolerancia a Fallos".

La Tolerancia a Fallos es una propiedad que permite al computador continuar con su comportamiento esperado, a pesar de la aparición de ciertos tipos de fallo, que de otro modo forzarían al sistema a entrar en un estado de error.

Antes de examinar las técnicas que se utilizan para dotar a los computadores de esta propiedad, veremos el concepto de fallo y los tipos

que se pueden presentar.

1.1.2. Tipos de fallo.

Un fallo es una condición anormal que se presenta durante la operación de un computador, cuya manifestación puede provocar la ejecución incorrecta de los algoritmos especificados (error).

Distinguiremos dos tipos de fallo: Fallos físicos, causados por fenómenos naturales adversos y fallos humanos, que resultan de las equivocaciones del hombre (por ejemplo, malas especificaciones, mala interacción hombre-máquina, etc.). Estos son los mas difíciles de corregir automáticamente. Nos centraremos en los fallos físicos, que están causados principalmente por tres fenómenos que afectan a la parte física del sistema:

-Fallos permanentes de componentes físicos.

-Malfunciones temporales de componentes físicos.

-Interferencias externas.

Podemos clasificar los fallos físicos según tres conceptos: Duración (transitorios-permanentes), extensión (locales-distribuidos) y valor (determinados-indeterminados).

Los fallos transitorios son de duración limitada. Su caracterización incluye una duración máxima, por encima de la cual pueden interpretarse como permanentes. Los fallos permanentes están causados por fallos irreversibles en los componentes.

La extensión de un fallo describe el número de variables lógicas que resultan afectadas simultáneamente debido a dicho fallo. Los fallos locales afectan a una sola variable lógica, mientras que los distribuidos afectan a dos o mas variables, a un módulo o incluso a todo el sistema.

Un fallo es determinado cuando las variables lógicas afectadas toman un valor constante mientras dura éste. El fallo es indeterminado cuando la variable lógica cambia de 0 a 1 o viceversa a lo largo de la duración del mismo.

Podemos redefinir el fallo físico del siguiente modo: Cambio de variables lógicas debido a la producción de una malfunción de algún componente físico. Por tanto, tendremos la siguiente secuencia de causa-efecto.

1.Una malfunción (física) provoca un fallo (cambio en variables lógicas).

2.El fallo proporciona entradas incorrectas al proceso de cálculo, con lo que puede provocar un error en las siguientes operaciones (aunque éstas estén libres de fallo).

Esta es la secuencia que debemos romper si queremos dotar a un computador de Tolerancia a Fallos.

1.1.3. Características de la tolerancia a Fallos.

Esta técnica aumenta la fiabilidad de un computador mediante el uso de redundancia protectiva. Sus características mas importantes son las siguientes:

-Las causas de fallo están presentes y pueden provocar errores durante el proceso de cálculo, pero sus efectos se contrarrestan automáticamente por medio de redundancia.

-Se hace posible así el funcionamiento correcto del sistema aún en presencia de ciertos fallos físicos, interferencias externas e incluso fallos humanos.

-Las partes redundantes (tanto componentes físicos como programas) o bien toman parte en el proceso de cálculo, o bien permanecen en espera,

listas para actuar automáticamente.

Evidentemente, la Tolerancia a Fallos no elimina totalmente la necesidad de utilizar componentes altamente fiables, pero lo que si logra es la reducción y localización de éstos en ciertos puntos clave del sistema.

1.1.4 Técnicas de Tolerancia a Fallos.

Desde el punto de vista funcional podemos distinguir dos técnicas de Tolerancia a Fallos: Enmascaramiento y Detección-recuperación.

1.1.4.1. Enmascaramiento de fallos.

Mediante el uso de esta técnica se emplea redundancia para asegurar que el efecto de un fallo queda totalmente contenido dentro de un módulo del sistema y no aparece en el exterior de éste. El fallo se cancela dentro del propio módulo y no aparecen a su salida síntomas ni efectos mientras que no se gaste toda la redundancia. Una vez agotada ésta, si se produce un nuevo fallo, el módulo podrá provocar un error. A esta técnica también se la conoce con el nombre de redundancia estática, puesto que desde fuera del módulo no pueden identificarse las funciones de detección y recuperación.

Los métodos mas utilizados de enmascaramiento han sido la duplicación de elementos individuales y la redundancia modular triple. Las réplicas redundantes de los elementos se mantienen continuamente conectadas y alimentadas, de modo que pueden proporcionar enmascaramiento instantánea y automáticamente. Mediante el uso de esta técnica protegemos al computador tanto contra los fallos transitorios como contra los permanentes. Sus mayores desventajas son las siguientes:

-Excesivo costo de la replicación masiva (3, 4, o mas veces el número original de componentes).

-Ausencia de aviso cuando se produce un fallo.

A pesar de ello, encuentra un gran campo de aplicación puesto que es de gran simplicidad conceptual, es transparente al usuario, y además tiene acción instantánea. Se utiliza sobre todo en módulos que no cuentan con ningún tipo de regularidad, y para proteger el llamado "nucleo duro".

1.1.4.2. Detección-recuperación de fallos.

Este método se caracteriza porque cuando se produce un fallo, la lógica de detección genera las señales de error que arrancan la recuperación de dicho fallo. La detección de fallos es una etapa fundamental, ya que la mejor técnica de recuperación puede ser únicamente tan buena como el detector que arranca su operación.

Para ejecutar la detección de fallos se utilizan tanto elementos físicos como programas de verificación que generan la indicación de fallo. Contamos fundamentalmente con dos tipos de detección: Simultánea y no simultánea.

La detección simultánea se ejecuta durante la operación normal del sistema. Así, la recuperación puede iniciarse antes de que los errores provoquen la ruptura de programas o la pérdida de datos. Se efectúa mediante elementos físicos especiales o mediante programas especiales que operan concurrentemente con los programas del sistema. Los métodos mas empleados para este tipo de detección son los siguientes:

-Códigos detectores de error.

-Duplicación y comparación.

-Circuitos especiales que verifican elementos críticos (reloj, alimentación, etc.).

-Circuitos lógicos autoverificados.

-Ejecución concurrente de programas.

-Rutinas de vigilancia intercaladas en el programa que se está ejecutando.

La detección no simultánea se caracteriza porque para efectuarla hay que interrumpir temporalmente la ejecución normal. Se efectúa mediante rutinas de verificación. Es muy útil hacer esta detección de un modo escalonado, es decir, comprobando primero una pequeña parte del sistema, y a continuación verificar el resto utilizando la parte cuyo buen funcionamiento ya se ha comprobado.

La recuperación comprende todas las acciones que se inician con una indicación de fallo por parte del detector, y concluye cuando se reanuda la operación normal del sistema. Pueden presentarse dos tipos de recuperación automática:

Recuperación total. El sistema vuelve al conjunto de condiciones existentes antes de que se produjera el fallo.

Recuperación degradada. Devuelve al sistema a un estado libre de fallo, pero con capacidad de cálculo reducida. Algunos elementos físicos quedan eliminados sin reemplazamiento (pérdida de programas y/o datos, ejecución de algunas funciones en tiempos mayores, etc.).

Desde otro punto de vista podemos distinguir también dos tipos de recuperación, la controlada físicamente y la controlada mediante programas. Los sistemas con recuperación controlada físicamente emplean elementos físicos específicos que recogen la indicación de fallo y ejecutan la recuperación.

Los sistemas con recuperación controlada por programa dependen de programas especiales que inician la recuperación cuando se ha detectado un fallo. La gran desventaja de estos sistemas es que (aparte de ser mas lentos) necesitan que los programas permanezcan operativos en presencia de fallos, pues de otro modo no podría iniciarse la recuperación.

Es evidente que el primer tipo de recuperación es mucho mas ventajoso, puesto que es independiente del funcionamiento de los programas inmediatamente después de ocurrido el fallo.

1.2 Objetivos de la Tesis.

Hemos visto que podemos utilizar diversas técnicas para lograr que un computador sea Tolerante a Fallos. Las mejores son aquellas que funcionan de un modo simultáneo, puesto que son capaces de proteger al sistema contra los fallos transitorios y permanentes; además son mucho mas rápidas que las demás. La técnica mas importante y ventajosa es la utilización de estrategias de detección y corrección de errores, aplicando la teoría de códigos al diseño de computadores.

Con el avance de la tecnología de semiconductores y la aparición de los circuitos de alto nivel de integración, el problema de la detección y corrección de errores toma un cariz muy especial. Esto es así porque, dejando a un lado el hecho de que en estos circuitos pueden producirse fallos de distinto tipo de los que se producen en componentes discretos o de nivel de integración bajo, el uso de éstos como subsistemas obliga a cambiar la partición del computador. Por tanto, también es necesario variar las estrategias de detección y corrección de errores utilizadas normalmente.

Por lo tanto, el objetivo fundamental de esta Tesis es investigar las estructuras de detección y corrección de errores que mas se adaptan al diseño de computadores Tolerantes a Fallos con lógica de alto nivel de integración, obteniendo los códigos detectores y correctores de error mas apropiados, adaptándolos a las estrategias planteadas y culminando en el diseño y construcción de algunos subsistemas Tolerantes a Fallos que incorporen dichas estructuras óptimas. En concreto, los pasos que se siguen son los siguientes:

- Estudio de la evolución de la tecnología de integración y tipificación de los fallos que se producen en los circuitos integrados.

- Estudio del estado actual de la teoría de códigos detectores y correctores de error, centrándonos en aquellos que son aplicables al diseño de computadores.

- Adaptación de dichos códigos al tipo de operaciones que se ejecutan en los computadores.

- Planteamiento de estrategias óptimas de detección y corrección de errores para los subsistemas mas importantes de los computadores.

- Diseño, realización y puesta a punto de algunos subsistemas Tolerantes a Fallos, aplicando los resultados obtenidos en la parte teórica.

1.3. Antecedentes.

En este apartado hacemos una breve revisión bibliográfica del tema que nos ocupa, viendo las fuentes que han servido de base a este trabajo.

Existen gran cantidad de publicaciones sobre Computadores Tolerantes a Fallos, siendo de especial interes las debidas a Avizienis (Avizienis 1971,1976,1977,1978) en las que se hace una revisión de las técnicas existentes, así como del estado del tema en aquellos momentos. Existen también otras publicaciones que tratan este tema de un modo general como (Ramamoorthy 1971), (Kime 1975,1978), (Stiffler 1976), (Bennets 1979), y otros.

La utilización de redundancia modular para el diseño de computadores Tolerantes a Fallos se aborda en las publicaciones: (Koczela 1971), (Rennels 1978), (Kameyana 1980) y (Ducasse 1980). También tratan el tema sin centrarse en ningún subsistema concreto las siguientes publicaciones: (Sedmak 1978,1980), (Cliff 1980) y (Goldberg 1980).

Históricamente la memoria ha sido la parte del computador mas susceptible de fallo (Goldberg 1974), y también en la que mejor pueden aplicarse las técnicas de Tolerancia a Fallos (sobre todo los códigos correctores de error) debido a su inherente regularidad y al gran número de elementos lógicos que la componen. Las publicaciones mas antiguas sobre el tema tratan de la protección de memorias de ferrita, aunque algunas de las técnicas obtenidas son también válidas para otras tecnologías. Mas recientemente se aborda de la protección de memorias de semiconductores.

Para esta protección pueden utilizarse técnicas de redundancia modular (Srinivasan 1971), matrices de conmutación (Szygenda 1971,1973), conmutación de elementos en espera (Hartwell 1978) y códigos correctores de error (Rao 1968).

Hsiao (Hsiao 1970) describe los códigos correctores de error utilizados en el IBM 7030 y en el IBM 360 mod. 18. Estos códigos son mejores que los convencionales de Hamming (Peterson 1961) tanto en velocidad de decodi-

ficación como en costo. Como la detección de un error se puede ejecutar mucho mas rápidamente que la corrección del mismo, Davida y Robinson (Davida 1970) desarrollan un método para, mediante un solo código corrector, ejecutar rápidamente la detección de error (entresacando del anterior un código detector), y si se produce éste, pasar al proceso de corrección. También Carter (Carter 1970) desarrolla un método de detección de errores dobles y corrección de errores simples, capaz además de detectar el fallo de los componentes del decodificador.

Para aumentar la efectividad de los códigos correctores de error simple, es técnica común organizar la memoria en rodajas de un solo bit (Stiffler 1978). Hsiao y Bossen (Bossen 1975) utilizan el método que llaman de los "cuadrados ortogonales latinos" para reconfigurar las líneas de dirección de bits individuales, asegurando de este modo que no se producirá mas de un error a la vez en una misma palabra de memoria (el método es equivalente a la división en rodajas de un bit). Carter y McCarty (Carter 1975,1976), (McCarty 1975) proponen un método de protección de memoria que combina el uso de códigos correctores de error simple con la técnica de conmutación de elementos en espera.

Cuando se organiza la memoria en rodajas de b bits, pueden utilizarse para su protección códigos sobre el campo $GF(2^b)$ (Peterson 1961). Esto propone Srinivasan (Srinivasan 1971) mejorando los resultados obtenidos por Bossen (Bossen 1970). Bossen y Chang (Bossen 1976,1978) hacen unas consideraciones sobre la capacidad de estos códigos. Fujiwara (Fujiwara 1979,1980) propone los códigos correctores de error en un solo bit, detectores de error en dos bits y detectores de error en una rodaja de b bits basandose en (Matsuzawa 1977).

Black (Black 1977) desarrolla un sistema de memoria altamente fiable utilizando el concepto de "borrado" para aumentar la capacidad de corrección del código utilizado. Otras publicaciones en el mismo sentido son: (Sundberg 1978,1979), (Walker 1979) y (Pradhan 1980).

Podemos aumentar aun mas la fiabilidad de una memoria protegida por

un código haciendo que los detectores estén formados por circuitos autoverificados. Entre las publicaciones en esta línea cabe destacar: (Carter 1971,1977), (Ashjaee 1976), (Smith 1977), (Wakerly 1974,1978) y (Marouf 1978). Thatte y Abraham (Thatte 1977) proponen un método de comprobación de memorias RAM, Hartwell (Hartwell 1978) propone una memoria Tolerante a Fallos para un sistema duplex, así como Cenker (Cenker 1979) propone un circuito integrado de memoria con capacidad interna de Tolerancia a Fallos. Cox y Carroll (Cox 1978) desarrollan modelos de fiabilidad para memorias con diversos tipos de protección (redundancia modular, elementos en espera, códigos, etc.). También trata del mismo tema (Ayache 1979). Levine y Meyers (Levine 1976) hacen un estudio de costo, tiempo y fiabilidad de la implantación de códigos correctores de error en la memoria de los computadores.

Aunque la unidad de proceso de un computador no es tan susceptible de fallo como la memoria, en algunos casos también es necesario protegerla contra éstos. La única alternativa frente a la replicación masiva para proteger estas unidades es el uso de códigos aritméticos correctores de error. Estos códigos fueron introducidos de un modo teórico por Peterson (Peterson 1961) y se han desarrollado a lo largo de las dos últimas décadas.

Rao (Rao 1968) propone la detección de errores en procesadores aritméticos mediante el uso de códigos AN, y Gaddes (Gaddes 1970) utiliza los códigos de resíduos para construir sumadores binarios con detección de error.

Avizienis (Avizienis 1971,1972) desarrolla una serie de criterios generales para determinar el costo y efectividad de la aplicación de códigos aritméticos detectores y correctores de error en los computadores digitales. Hace el estudio sobre todo para los códigos que llama de "bajo costo".

Vista la dificultad para trabajar con los códigos AN, Rao (Rao 1972) desarrolla unos subcódigos sistemáticos, a los que llama gAN, que permiten abordar de un modo mas cómodo el problema. Además, Neumann y Rao (Neumann 1973,1975) proponen unos códigos para la corrección de errores en

sumadores organizados en rodajas de b bits.

Pradhan (Pradhan 1974) presenta técnicas de diseño de sumadores rápidos y unidades aritméticas Tolerantes a Fallos basadas en el uso de códigos no binarios y de Reed-Muller. Chinal (Chinal 1975) analiza los sumadores en complemento a dígito para su aplicación en unidades aritméticas Tolerantes a Fallos. Rao y Reinheimer (Rao 1977) proponen una unidad aritmética modularizada protegida mediante códigos que llaman de "combinación", que aglutinan códigos de paridad con códigos de residuos. Por último, Wakerly en su libro (Wakerly 1978) esboza el diseño de una unidad de proceso con detección de errores basada en la aplicación de códigos de residuos.

Cabe citar igualmente una serie de publicaciones generales sobre teoría de códigos y también las que profundizan en algunos concretos, que son las siguientes: (Berlekamp 1968, 1980), (Lin 1970), (Hong 1972), (McWilliams 1977), (Bhargava 1978), (Devries 1979), y (Pradhan 1980).

Estas publicaciones, junto con las que se referencian a lo largo del texto, así como algunas mas que no se citan por no tener una relación tan directa con el tema, han servido de base para la realización de este trabajo.

CAPITULO II.

CONSIDERACIONES TECNOLOGICAS.

Pretendemos en este capítulo ofrecer una revisión de la evolución y tendencias de la tecnología de semiconductores, así como un estudio de los fallos que pueden producirse mas frecuentemente en los circuitos integrados. Ambos aspectos nos servirán de base a la hora de plantear las estrategias de detección y corrección de errores en computadores.

2.1. Evolución de la tecnología de semiconductores.

La tecnología de semiconductores ha sufrido en los últimos años un crecimiento inusitado. La complejidad de los circuitos integrados se ha incrementado en varios órdenes de magnitud, mientras que el costo por puerta ha descendido de un modo similar. Debido a estos avances de la tecnología, el costo de los computadores ha caido dramáticamente. Ello hace que dichos avances tengan una fuerte influencia sobre la arquitectura de estos sistemas, así como sobre cada una de sus partes (procesador, memoria, etc.).

2.1.1. Desarrollo histórico.

Podemos expresar la complejidad de los circuitos integrados de semiconductores en términos del número de sus componentes (transistores) (Bhandarkar 1979).

Atendiendo al nivel de integración (número de transistores por circuito integrado), se distinguen tres generaciones llamadas SSI, MSI y LSI, además de una cuarta en cuyo comienzo estamos, que se conoce por el nombre de VLSI.

La primera generación de circuitos integrados, SSI (integración en pequeña escala), comenzó a principios de la década de los 60 y utilizaba

principalmente tecnología bipolar. Cada circuito integrado contenía entre 15 y 100 transistores. Estos circuitos integrados eran (y siguen siendo) muy útiles en la mayoría de los sistemas. Consisten principalmente en puertas lógicas y biestables.

Alrededor de 1965 comenzó la segunda generación de circuitos integrados, MSI (integración a escala media). Esta generación se caracterizó porque cada circuito integrado contenía entre 100 y 1000 transistores (dependiendo de la función y de la fecha). Los principales bloques que se realizaron en esta segunda generación fueron contadores, decodificadores, multiplexores, operadores, registros, etc. Es durante esta generación cuando surge la tecnología MOS, con la que se logra una densidad de integración apreciablemente mayor.

Durante la tercera generación, LSI (alto nivel de integración), cuyo comienzo puede situarse al principio de la década de los 70, se integran en un solo circuito sistemas que constan de entre 1000 y 10000 transistores. Es el momento en que aparecen los microprocesadores y las primeras memorias de semiconductores. Durante esta generación impera la tecnología MOS, aunque también con tecnología bipolar se fabrican circuitos altamente integrados, tales como los microprocesadores de rodajas (que contienen del orden de 2500 transistores), secuenciadores de microprograma, etc. Se le da un gran impulso a las memorias MOS.

En nuestros días estamos inmersos en una cuarta generación de circuitos integrados, VLSI, en la que se construyen sistemas con mas de 10000 transistores. Ya en 1979 eran realidad las memorias de acceso aleatorio (RAM) con una capacidad de 64 K-bits y los microprocesadores de 16 bits. Las memorias de 256 K-bits y los microprocesadores de 32 bits, con los que se está trabajando actualmente, ya tocan los límites de la resolución óptica (Holton 1980). Será necesario avanzar en las técnicas de exposición con rayos X y con electrones para poder continuar el avance de la tecnología de semiconductores.

En la figura 2.1 se muestra el desarrollo del crecimiento en com-

plejidad de los circuitos integrados. Los puntos indican los circuitos mas complejos existentes en el mercado en el momento considerado. Se deduce de la figura que los progresos en la fabricación de circuitos integrados cada vez mas complejos aumentan de un modo exponencial (la pendiente de la curva aproximadamente se duplica cada año).

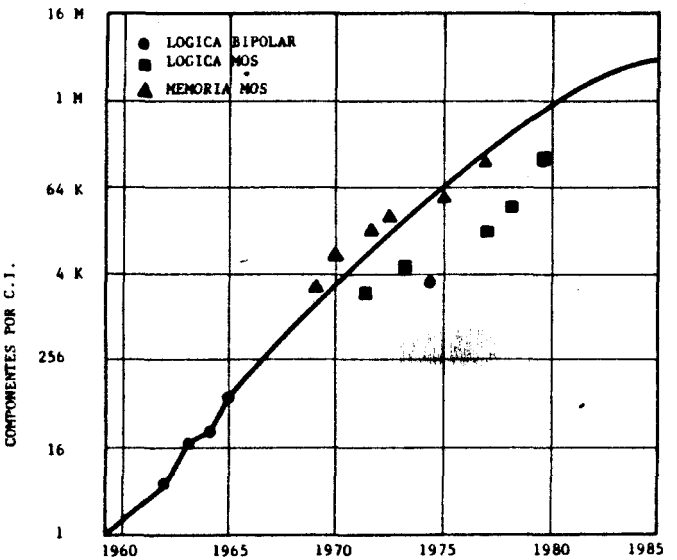


Figura 2.1

La disminución prevista en esta curva del crecimiento de la pendiente se debe al gran problema de establecer que producto debe fabricarse para que sea rentable (en número de unidades fabricadas) y que se acerque al límite tecnológico.

Este mismo problema se presentó en 1965 al tratar de dividir los sistemas digitales en bloques tan complejos como lo permitía la tecnología de la época. Se presentaron dos problemas:

-El número de interconexiones entre circuitos aumentaba de un modo tan rápido con la complejidad del circuito, que pronto se sobrepasaban las posibilidades de encapsulado del momento.

-Los bloques tendían a ser únicos, cosa que provocaba la aparición de muchos diferentes, perjudicando así el abaratamiento del producto.

La crisis de definición de producto limitó la complejidad de los circuitos integrados hasta 1968. Esta crisis se superó por dos razones: El desarrollo del calculador y la aparición de las memorias de semiconductores.

El calculador era un sistema digital bastante simple que podía dividirse en unos cuatro circuitos integrados de 40 patas, haciendo tratable el problema de las interconexiones. Además se fabricaron en grandes cantidades, redundando en el abaratamiento de los componentes y justificando los costos de diseño.

Con los circuitos integrados de memoria, debido a su función universal se pudo alcanzar el máximo nivel de integración disponible. Se redujo el número de conexiones con el exterior incorporando el decodificador de dirección dentro del propio circuito, contribuyendo así a su abaratamiento, que la hizo competitiva con el resto de las tecnologías de memoria existentes.

Consecuencia de los circuitos integrados para el calculador y los de memoria fue el microprocesador, cuya aparición extendió ampliamente el uso de los circuitos integrados. Por su arquitectura de propósito general fue (y sigue siendo) el componente idóneo para gran cantidad de aplicaciones, variando simplemente los programas para cada una de ellas.

Así pues, durante la decada de los 70, la industria de semiconductores se ha dedicado a desarrollar circuitos integrados de memoria cada vez mas complejos, así como microprocesadores y sus periféricos correspondientes. De este modo hemos llegado a tener en el mercado memorias de semicon-

ductores con una capacidad de 64 K-bits y microprocesadores de 16 bits, que contienen del orden de 68000 transistores, estando ya a un nivel experimental muy avanzado las memorias de 256 K-bytes y los microprocesadores de 32 bits (con alguno ya en el mercado).

Estos enormes progresos en la escala de integración han sido posibles gracias al uso de tecnología MOS. Con tecnología bipolar, debido fundamentalmente a limitaciones de disipación de potencia, se ha llegado solamente a la cuarta parte de la densidad de integración de la tecnología MOS (Bhandarkar 1979). Aún así, gracias a su indudable mayor velocidad, los circuitos integrados LSI de tecnología bipolar encuentran aplicación en la construcción de gran número de sistemas.

2.1.2. Previsiones futuras.

Hemos visto que hasta hoy el progreso de la tecnología de semiconductores ha sido extremadamente rápido. Todos los aspectos del proceso de datos se han beneficiado de las grandes ventajas logradas en costo, fiabilidad y capacidad de los componentes electrónicos. Es posible hacer una estimación de la evolución de la tecnología de circuitos integrados en un futuro, y prever una continuación de este rápido progreso (Keyes 1979). Ahora bien, esta estimación debe hacerse teniendo en cuenta tanto los fenómenos conocidos que se producen en los semiconductores y las leyes físicas, como la rentabilidad obtenida con este progreso.

En la figura 2.2 se muestra una estimación de las dimensiones del dado de silicio, y de la dimensión mínima que es posible reproducir con el proceso litográfico.

La figura 2.3 muestra el número de elementos resolubles por circuito integrado, obtenido de la figura 2.2. Igualmente se hace una estimación del número de elementos necesario para construir una puerta lógica, así como un bit de memoria con su cableado asociado. Al progreso de las escalas de integración contribuyen tanto el aumento de tamaño del dado como la disminución de la dimensión mínima (aumento del número de elementos resolu-

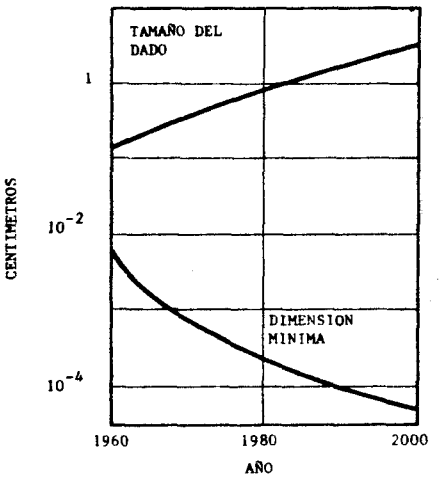


Figura 2.2

bles).

Para alcanzar los progresos que se muestran en las dos figuras anteriores se necesitarán contínuos programas de desarrollo y, por tanto, sólo tendrán lugar si pueden utilizarse económicamente para construir sistemas funcionales.

Existen algunos factores que probablememte limitarán el desarrollo de circuitos altamente integrados. Estos incluyen los siguientes:

-Electromigración. Este fenómeno, que veremos como uno de los modos de fallo de los circuitos integrados, provoca un problema importante cuando disminuimos el grosor de las pistas de conexión interna (por la miniaturización). El problema podrá paliarse en parte mediante el desarrollo de mejores aleaciones.

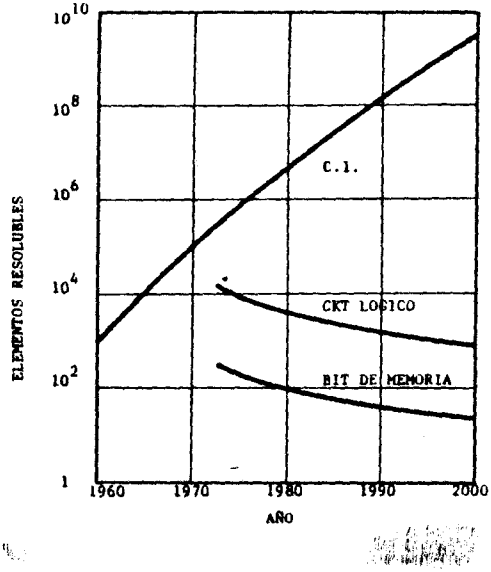


Figura 2.3

-Perforación de dielectricos. En dispositivos con pistas mas estrechas que una micra los campos eléctricos pueden producir la rotura de dieléctricos, ralentizándose por este problema el desarrollo de los circuitos integrados. Para poder explotar totalmente la máxima capacidad prevista para los procesos litográficos (haz de electrones, rayos X, etc.) habrá que pensar en el desarrollo de algún material semiconductor distinto del silicio. Podemos estimar razonablemente entonces que en el año 2000 serán comunes las pistas de conexión con un ancho de 0,5 micras.

-Otro límite potencial al progreso de la tecnología de semiconductores es el incremento de la resistencia ohmica de las pistas de conexión interna. El incremento de longitud de dichas pistas en un circuito integrado, debido al incremento del número de circuitos elementales, llegará a convertirse en un grave problema (Heller 1977). La dependencia entre la longitud de conexiones y el número de circuitos elementales por circuito

integrado se muestra en la figura 2.4.

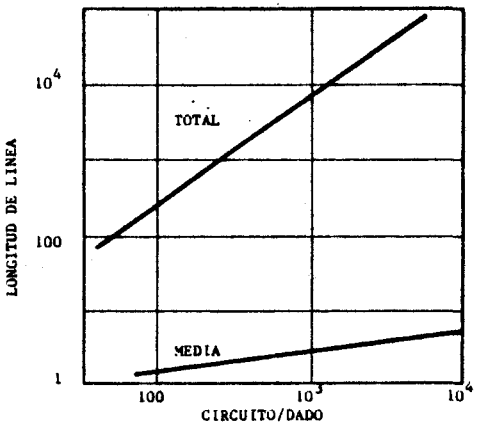


Figura 2.4

-Complejidad de cableado. Este problema constituye una limitación mas inmediata al progreso de la tecnología. El hecho de incrementar el número de circuitos elementales en un circuito integrado, aumenta el número de conexiones entre el dado y su substrato. Fijandonos en la dificultad que se encuentra actualmente para efectuar 100 de estas conexiones, podemos afirmar que el número de conexiones necesarias en el futuro (se estima que 10000 circuitos lógicos necesitan 1000 conexiones para grandes sistemas) serán aún mas difíciles de hacer. La figura 2.5 es una extrapolación de los resultados obtenidos por Landman y Russo (Landman 1971) para un pequeño número de circuitos, pero suponemos que el número de conexiones por circuito integrado no excederá de 1000. Incluso se reducirá bastante integrando un subsistema funcional completo.

En la figura 2.6 se estima la velocidad de crecimiento prevista del número de bits por circuito integrado para memorias y del número de circui-

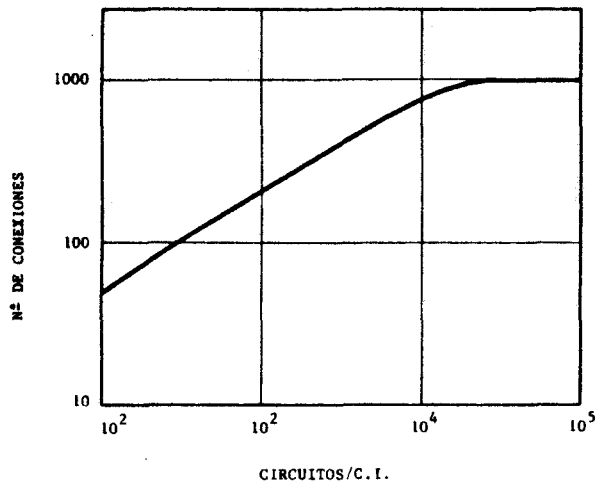


Figura 2.5

tos lógicos por circuito integrado para lógica aleatoria.

En esta figura se observa un menor crecimiento para la lógica que para la memoria, y además se aprecia una disminución de la pendiente a partir de 1990. Esto es debido a la capacidad tecnológica para producir dados mayores.

De todo lo anterior podemos deducir que, si las únicas restricciones al crecimiento de la escala de integración son las tecnológicas, a finales de este siglo dispondremos de circuitos integrados de memoria capaces de almacenar 10^7 bits, y circuitos integrados de lógica compuestos por 10^6 transistores. Existe sin embargo un problema fundamental que se espera se resuelva en los próximos años, y es el de saber "que integrar en un solo circuito para que éste pueda cubrir los gastos de desarrollo".

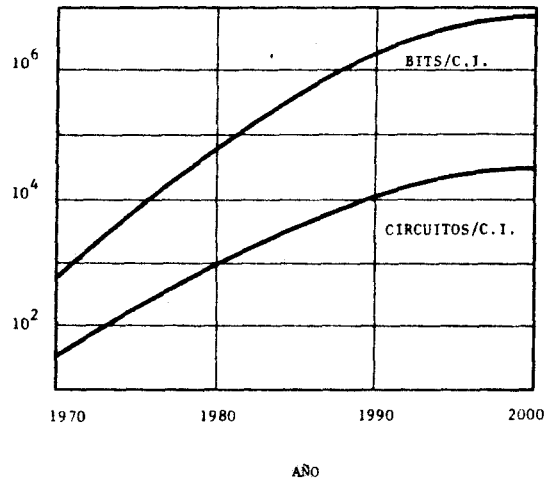


Figura 2.6

2.1.3. Estrategias de integración.

Una última consideración a hacer en cuanto a la tecnología es la estrategia actual de integración, que va a condicionar la arquitectura de nuestros sistemas. Vamos a considerar los dos tipos generales de circuitos integrados digitales que se encuentran en el mercado: Memorias y lógica aleatoria.

-Memorias. Existen en el mercado circuitos integrados de memoria con diversas organizaciones (ED. 1980). Estas podemos agruparlas en dos tipos: Accesibles bit a bit y accesibles por grupos de "b" bits. Por ejemplo, las actuales memorias de 64 K-bits están organizadas en 8Kx8, es decir, que contienen 8 K-palabras de 8 bits, o bien mas frecuentemente en 64Kx1, es decir, que contienen 64 K-palabras de 1 bit. La elección de un tipo u otro de organización, debido a los distintos tipos de fallo que implican, será de gran importancia a la hora de plantear las estrategias co-

rrespondientes de detección y corrección de errores.

-Lógica aleatoria. Del mismo modo que en las memorias, en los circuitos integrados de lógica aleatoria existen varios tipos de particiones, desde la integración de un sistema completo como suele hacerse en tecnología MOS (vease el caso de los microprocesadores) hasta la integración de partes mas o menos grandes de un sistema como se hace en tecnología bipolar (debido normalmente a problemas de disipación de potencia). En tecnología bipolar la partición del sistema suele hacerse en rodajas verticales iguales de un ancho determinado. La elección de uno u otro tipo de partición también influirá mucho en las estrategias que utilicemos en la protección contra errores.

2.2. Fallos mas frecuentes en circuitos integrados.

Tratamos en este apartado de tipificar los mecanismos de fallo mas frecuentes en los circuitos integrados (especialmente en los de alto nivel de integración) para concluir como van a influir los efectos de dichos fallos en la elección de las estrategias de recuperación de los mismos.

Vimos en la introducción que podemos agrupar los fallos en dos tipos principales: Fallos permanentes y fallos transitorios.

Los fallos permanentes son aquellos en que un mecanismo "físico" provoca alguna anomalía en el interior del circuito, resultando un daño permanente en el mismo.

Los fallos transitorios son aquellos en que el fenómeno inductor de los mismos provoca un cambio en algún estado lógico o en algún dato almacenado, pero sin causar un daño permanente en el circuito. Un ejemplo de este tipo de fallos son los provocados por las partículas "α" (May 1979).

2.2.1. Fallos permanentes.

Analizaremos los fallos permanentes que se producen en los circuitos integrados, tanto en tecnología bipolar como en tecnología MOS.

2.2.1.1. Tecnología bipolar.

Un circuito integrado construido con tecnología bipolar podemos dividirlo en dos capas. Una primera capa que contiene las células semiconductoras básicas, y una segunda compuesta por las conexiones entre células. Los fallos permanentes en la primera capa suelen ser muy poco frecuentes, y normalmente dan lugar a errores simples, mientras que los fallos en la capa de conexiones son los que se producen mas frecuentemente y dan lugar a errores distribuidos.

2.2.1.1.1. Fallos en las conexiones internas.

Estos fallos se deben principalmente al fenómeno llamado de "electromigración" (Sahni 1970). Dicho fenómeno consiste en un movimiento de masa provocado por dos causas fundamentales: Grandes densidades de corriente en las pistas metálicas de conexión, y temperaturas elevadas en el interior del circuito integrado debido a la disipación de potencia. Los fallos en las conexiones pueden ser de dos tipos:

a)Fallos en la pista de aluminio. Este modo de fallo, que provoca un "circuito abierto" (figura 2.7) ocurre en los siguientes lugares:

Figura 2.7

1)Aquellos en que las pistas son mas estrechas y/o mas finas que

los valores nominales correspondientes. El hecho de ser mas finas ocurre debido a variaciones del grosor de la película metálica depositada durante el proceso de fabricación, y el hecho de aparecer mas estrechas se produce por un efecto de pérdida de definición de los bordes en el proceso litográfico. Los lugares donde coinciden estas dos variaciones constituyen lugares potenciales de fallo.

2)Otros lugares de fallo potencial son aquellos puntos donde no hay uniformidad en el tamaño de grano en la película de aluminio. El arrastre de material es directamente proporcional al gradiente de la composición de la película (Sahni 1970).

3)Debido a que el fenómeno de electromigración también se ve favorecido por el gradiente de temperatura, otros lugares potenciales de fallo son los puntos en que la pista pasa sobre algún componente que disipe mucha potencia (p. ej. resistencia).

b)Fallos en el contacto aluminio-silicio. Como el transporte de átomos metálicos debido a la electromigración se produce en la dirección del flujo de electrones, en resistencias por las que circule mucha corriente se produce una acumulación de aluminio en el terminal negativo, así como una despoblación en el terminal positivo (figura 2.8).

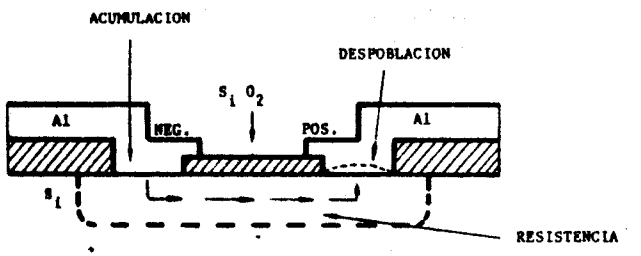


Figura 2.8

Cuando los átomos de aluminio transportados llegan cerca del termi-

nal negativo de la resistencia, éstos se presentan en la interfase aluminio-silicio. Como no pueden ir mas allá, se produce una acumulación de aluminio en este terminal. Por el contrario, en el terminal positivo la electromigración arrastra los átomos de aluminio lejos del contacto, y como no hay átomos de aluminio antes de dicho contacto para reemplazar los perdidos, se provoca una despoblación del contacto que, con el tiempo, se convertirá en un circuito abierto.

Estos fallos, que son los mas comunes en circuitos integrados construidos con tecnología bipolar, ocurren de un modo "natural", es decir, sin forzar las condiciones de utilización del circuito (tension de alimentación, temperatura, etc.). Si se fuerzan estas condiciones, pueden ocurrir además otros tipos de fallo, tales como cortocircuitos, perforación de uniones, etc.

Así pues, en condiciones normales de utilización, los fallos mas frecuentes en tecnología bipolar consisten en circuitos abiertos en las conexiones internas del circuito integrado, que producen errores distribuidos, es decir, que afectan a varias variables de salida.

2.2.1.2. Tecnonogía MOS.

Los circuitos integrados construidos con tecnología MOS podemos dividirlos en tres niveles de conexión interna (Galiay 1980). 1) Un nivel inferior de conexiones hechas por difusión sobre un substrato aislante. 2) Un nivel superior de conexiones hechas por metalización. 3) Un nivel intermedio de óxido que aisla los dos niveles anteriores y presenta dos tipos de discontinuidades: a)"orificios" que permiten el contacto entre metalizaciones del nivel superior y difusiones del nivel inferior, y b)"ventanas" que corresponden a las puertas de los transistores.

La realización de las difusiones y metalizaciones necesita un enmascaramiento selectivo de regiones muy precisas en la superficie del circuito integrado. Un fallo inherente a este proceso consiste en difundir o metalizar regiones que no deberían difundirse o metalizarse, o viceversa.

De este trabajo experimental se obtuvieron las conclusiones siguientes:

a)Los fallos se distribuyen aleatoriamente, y ninguna zona del circuito integrado es mas susceptible de fallo que las demás.

b)Los fallos consisten principalmente en cortocircuitos y circuitos abiertos a nivel de metalización o de difusión.

En el 10% de los casos se estableció claramente la existencia de un fallo lógico (error en la salida), pero no pudo observarse ningún fallo físico. En el 15% de los casos, el circuito presentaba una imperfección que afectaba a la mayor parte de las funciones del circuito integrado (fallo catastrófico).

Observamos entonces que los fallos permanentes en circuitos integrados construidos con tecnología MOS consisten también en circuitos abiertos y además en cortocircuitos. También se deduce que la mayor parte de los fallos producen errores distribuidos. Normalmente una zona del circuito integrado quedará afectada por el fallo, mientras que el resto del circuito seguirá funcionando correctamente.

2.2.2. Fallos transitorios.

Uno de los principales agentes productores de fallos transitorios en circuitos altamente integrados es el fenómeno de emisión de partículas "α" (McPartland 1980). En casi todos los materiales existen pequeñas cantidades de uranio y torio. En concreto, en los materiales utilizados para encapsular los circuitos integrados existen en cantidades del orden de 50 partes por millón (ppm). Estos dos materiales radiactivos emiten partículas "α" con energías de hasta 8,78 MeV.

Al moverse por el silicio una partícula "α", ésta pierde energía, debido a lo cual, se crean pares electrón-hueco que se separan de modo que las regiones "n" atrapan los electrones y las regiones "p" los huecos antes

de que puedan recombinarse. Este mecanismo de aumento de carga en algunas regiones puede provocar la pérdida de datos almacenados, o bien el cambio de alguna variable lógica (figura 2.9).

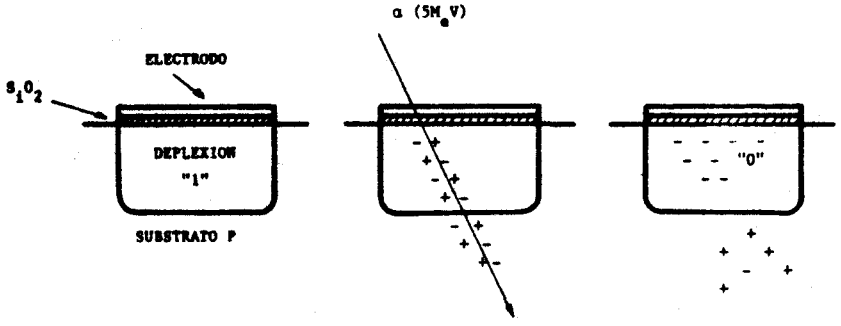


Figura 2.9

Debido a que en los dispositivos integrados a gran escala las cargas almacenadas son muy pequeñas, la cantidad de fallos inducidos por partículas "α" se hace muy importante. Este tipo de fallo se da principalmente en las memorias dinámicas (MOS).

Los fallos transitorios debidos a partículas "α" están siendo muy estudiados (May 1979), (Peeples 1980), tanto la relación del flujo de partículas (para distintos materiales de encapsulado) con el número de fallos producidos, como su relación con otros factores como temperatura, tensión de alimentación, etc. La conclusión que puede sacarse de estos estudios es que cada vez se hace mas importante el efecto de las partículas "α". Esto es así porque, como cada vez se logra una mayor miniaturización de las células elementales, una sola partícula "α" afecta a un numero mayor de éstas. Debido igualmente a la miniaturización, la carga almacenada en estas células es cada vez menor, así que se pueden producir fallos con partículas " " de niveles energéticos menores.

Igualmente hay que empezar a tener en cuenta otros agentes inductores de fallo como son:

-Rayos Cósmicos, que aunque su flujo a nivel del mar no es significativo, hay que tenerlos en cuenta para sistemas funcionando en satélites artificiales (Pickel 1978).

-Ruido de generación-recombinación. Los efectos de este agente comienzan a hacerse importantes debido a la reducción de las dimensiones, así como de las tensiones de alimentación (Keyes 1975).

2.2.3. Tipificación de fallos.

Como resumen de lo anterior podemos afirmar que los fallos mas probables en circuitos integrados a alto nivel consisten principalmente en circuitos abiertos y cortocircuitos (mas probables los circuitos abiertos en tecnología bipolar) de las metalizaciones de conexión interna del circuito integrado, y que éstos en su mayor parte dan lugar a errores distribuidos que afectan a mas de una variable lógica de salida. Esto, junto con la partición elegida para el sistema teniendo en cuenta la escala de integración, influye fuertemente en la elección de las estrategias de Tolerancia a Fallos que se utilizan, así como de la propia arquitectura del sistema.

En cuanto a los fallos transitorios, pódemos concluir que en su gran mayoría están producidos por la emisión de partículas "α". Tratar de evitarlos reduciendo la cantidad de material radiactivo contenida en los materiales de encapsulado no resulta rentable. El mejor método para proteger un sistema frente a este tipo de fallos es el uso de algún tipo de detección y corrección simultánea. De tales métodos, el mejor es la utilización de códigos detectores y correctores de error, tal como vemos en capítulos posteriores.

CAPITULO III.

CODIGOS DETECTORES Y CORRECTORES DE ERROR.

En este capítulo hacemos una recopilación, sistematización y puesta al día de algunos códigos detectores y correctores de error dispersos en la literatura existente sobre el tema. Además, este capítulo sirve de base a los desarrollos que se emprenden en los capítulos siguientes.

Después de establecer una clasificación de los diferentes tipos de código existentes, nos centraremos en los códigos de bloques lineales y en los códigos aritméticos, ya que son los mas apropiados para su aplicación en el diseño de computadores Tolerantes a Fallos.

3.1. Introducción.

Como ya hemos visto, los sistemas son susceptibles de fallo. Para protegerlos, podemos codificar su salida de modo que, durante su operación normal (libre de fallos), dicha salida tome solamente un subconjunto de todos los valores posibles. Como durante la operación normal del sistema no aparecen en la salida valores que no pertenezcan al subconjunto antes citado, la aparición de uno de éstos deberá interpretarse como un error. A continuación estudiamos los métodos de codificación para poder detectar y corregir los errores.

3.2. Definición.

Un código detector y corrector de errores es un subconjunto C del universo U de todos los vectores. Elegimos C de tal modo que los fallos que ocurran mas frecuentemente, hagan variar los vectores de salida X C cambiándolos en otros X^* que no estén en C, $X^* \notin C$.

Llamaremos palabra código a cualquier vector X tal que $X \in C$. Una palabra que no sea del código será un vector X^* del conjunto U-C, es decir, $X^* \in U-C$. Así pues, si un fallo provoca un cambio del vector $X \in C$ en un vector $X^* \in U-C$, se habrá producido un error detectable (y a veces corregible). En cambio, si el fallo provoca que el vector $X \in C$ se transforme en otro X también perteneciente al subconjunto C, el error producido permanecerá indetectado, puesto que se ha pasado de una palabra código a otra de las mismas características.

Mas adelante introduciremos modelos de error para relacionar los fallos con los errores que éstos producen. Estructuraremos nuestros códigos detectores y correctores de error de modo que los errores mas probables (dentro de nuestro modelo) no queden indetectados, es decir, que produzcan palabras no pertenecientes al código.

Para distinguir en la salida las palabras código de las que no lo son, hay que interpretar ésta mediante un decodificador. Este puede simplemente notificar que se ha producido un error (cuando la palabra no es del código). De este modo obtenemos la función de detección de errores. Cuando el decodificador también es capaz de asociar la palabra errónea con la palabra código que tendríamos si no se hubiese producido error, se está ejecutando además la función de corrección de errores.

El hecho de detectar o corregir errores con un código determinado depende del decodificador, pues como veremos, un mismo código puede tener capacidad, por ejemplo, de detectar errores dobles o bien corregir errores simples, dependiendo del decodificador empleado.

3.3. Clasificación.

Los códigos detectores y correctores de error han sido muy estudiados y se han utilizado mucho desde su aparición (sobre todo para proteger la transmisión de datos). A continuación daremos una clasificación de los mismos citando los tipos de código mas conocidos.

Podemos clasificar los códigos atendiendo al modo en que se concatenan los bits redundantes con los de información en dos grandes grupos: Códigos de Bloque y Códigos de Arbol. Un código de bloque es aquel en que la información se segmenta en bloques de un tamaño determinado, que se codifican para formar la palabra código. En un código de arbol no se puede distinguir claramente a que símbolos de información corresponden unos símbolos de control determinados. No se segmenta la información.

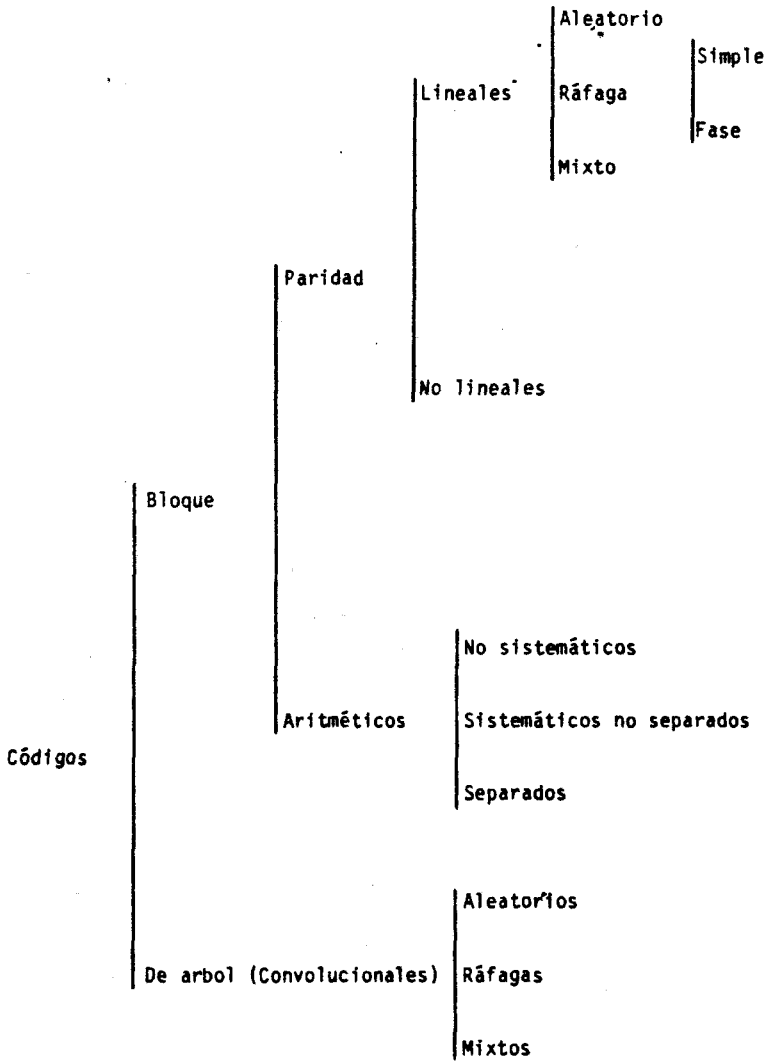
Dentro de los códigos de bloque, podemos distinguir entre códigos de tipo paridad y códigos àritméticos, diferenciandose además de en su modo de codificación y decodificación, en sus respectivos campos de aplicación. Dentro de los de tipo paridad distinguimos entre lineales y no lineales, pudiendo ser los primeros dependiendo del tipo de error que corrigen: aleatorio, de ráfagas y mixto.

Encontramos entre los códigos que corrigen errores aleatorios los de Hamming como sus representantes mas conocidos. Se utilizan sobre todo para corregir errores simples. Para corregir mas de un error, suelen utilizarse los códigos BCH (Bose Chaundri Hocquenghem), tanto binarios (Reed Muller) como no binarios (Reed Solomon, Justener).

En cuanto a los códigos que corrigen ráfagas, se pueden dividir en dos clases: Los que corrigen ráfagas simples y los que corrigen o controlan fases de ráfagas. Ambos tipos se utilizan para controlar canales de transmisión. Entre los primeros podemos citar los códigos FIRE y los INTERLACED, llamados así por su forma de decodificación. Entre los que controlan fases de ráfagas tenemos como representante el código BURTON.

Hay otra clase de códigos lineales que se utilizan cuando el tipo de error no es muy concreto, estos son los códigos mixtos, y entre ellos se pueden citar los códigos producto y los códigos concatenados.

Los códigos no lineales se utilizan para conseguir el máximo número de palabras código para una longitud determinada. Entre ellos están los códigos de Hadamard, de Golay y de Nodstrom-Robinson.


Aleatorio
Simple
Lineales
Ráfaga
Fase
Mixto
Paridad
No lineales
Bloque
No sistemáticos
Aritméticos
Sistemáticos no separados
Códigos
Separados
Aleatorios
De arbol (Convolucionales)
Ráfagas
Mixtos

Los códigos aritméticos podemos clasificarlos en tres grupos: No sistemáticos, cuyos representantes son los AN. Sistemáticos no separados, que tienen como principal representante los códigos gAN. Separados, cuyo ejemplo mas importante son los códigos de residuos.

Dentro de la categoría de los códigos de arbol citaremos solamente el grupo mas importante, que son los Convolucionales. Sus principales características son su gran complejidad de implantación y su gran costo. En contrapartida, son mas fiables. Este tipo de código se utiliza fundamentalmente para corregir ráfagas de error en comunicaciones. Entre los que corrigen errores aleatorios están los códigos WYNNER-ASH y los SELF-ORTOGONAL. Para ráfagas están los IDAWARE, y hay códigos mixtos como los de GALAGER.

Existen además otros códigos que no pueden encuadrarse de un modo exacto en las categorías anteriores, y que sin embargo son muy utilizados. Entre ellos podemos citar los códigos "m entre n" y los de residuo cuadrático. En la tabla se resumen las ideas citadas anteriormente.

A continuación profundizaremos en los códigos que han demostrado ser mas útiles en su aplicación al diseño de computadores Tolerantes a Fallos.

3.4. Códigos de bloques lineales.

Con este tipo de códigos, la información se codifica en vectores cuyos símbolos pertenecen a un campo finito GF(p) con p elementos. Existen códigos lineales para cualquier valor de p, potencia de un número primo. Seguidamente tratamos de los códigos en los que p es de la forma 2^n.

En un espacio vectorial sobre GF(p) existen p^n vectores diferentes, de la forma $X = (X_1, X_2, \ldots, X_n)^T$ con $X_i \in GF(p)$. Sin embargo, solamente son palabras código un subconjunto C que contiene p^k vectores (n>k). A este subconjunto lo llamamos código (n,k).

3.4.1. Definición.

En el campo GF(p) están definidas dos operaciones, la suma (denotada por +) y la multiplicación (denotada por $^{\circ}$). Para p = 2, estas operaciones son la suma módulo 2 ("O" exclusivo) y el "Y" lógico binario.

En el código, que es un subespacio vectorial, también están definidas dos operaciones, la suma de vectores y el producto por un escalar (perteneciente a GF(p)), del siguiente modo:

$$X + Y = (X_1 \oplus Y_1,\ X_2 \oplus Y_2,\ \ldots,\ X_n \oplus Y_n)$$

$$CX = (CX_1,\ CX_2,\ \ldots,\ CX_n)$$

A partir de este preambulo, podemos dar una definición formal de los códigos lineales:

Un código lineal es un subespacio del espacio vectorial formado por todos los vectores de dimensión n sobre GF(p).

Podríamos describir un código lineal por extensión, es decir, mediante una lista exhaustiva de todos los vectores que lo forman, pero existe un modo mas compacto de hacerlo; podemos describirlo en términos de matrices.

Cualquier base de un código lineal (que es un espacio vectorial), puede considerarse como las filas de una matriz G, que llamamos matriz generadora. Así, un vector es una palabra código si y solo si es una combinación lineal de las filas de la matriz G.

Otra alternativa para la descripción de un código mediante matrices es la siguiente. Sea H una matriz de rango n-k. Un vector es una palabra código si y solo si se verifica que:

$$HX^t = 0 \qquad (1)$$

donde X^t es el vector traspuesto de X y 0 es el vector nulo.

Esto quiere decir, que se verifica:

$$\sum_j h_{ij} a_j = 0$$

Por lo tanto, los componentes del vector X deben satisfacer un conjunto de n-k ecuaciones linealmente independientes para que X sea una palabra código. A la matriz H la llamamos matriz de paridad.

Existe una relación entre la matriz de paridad H y la matriz generadora G. Como las filas de G son palabras código (forman una base del código), el producto de H por la traspuesta de cada una de ellas debe ser nulo. Por lo tanto, se verifica que:

$$HG^t = 0$$

3.4.2. Propiedades de un código lineal.

1.- Cuando el código es sistemático, la matriz H toma la forma $H = (A \circ I_{n-k})$, siendo A una matriz de dimensiones (n-k) x k e I_{n-k} la matriz identidad de dimensiones (n-k) x (n-k). Así vemos que existen 2^k palabras código que satisfacen la ecuación (1).

Cuando H tiene la forma citada, las palabras código tienen separados los símbolos de información de los símbolos redundantes.

$$X = (X_1, X_2, \dots, X_k, X_{k+1}, \dots, X_n)$$

Símbolos de información — Símbolos redundantes

2.- La generación de las palabras código se hace del siguiente modo:

Sea $u = (u_1, u_2, \dots, u_k)$ la información que queremos codificar. Por

la definición de matriz generadora G, la palabra código resultante será:

$$X = uG.$$

Si H toma la forma citada en la propiedad 1, como $HG^t = 0$, se puede deducir facilmente que G tendrá la forma:

$$G = (I_k^p - A^t)$$

3.- Los parámetros principales de un código lineal son su longitud y su dimensión. Estos le caracterizan.

La palabra código $X = (X_1, X_2, \ldots, X_n)$ se dice que tiene longitud n. Por tanto, el código correspondiente tiene longitud n.

Si H tiene n-k filas linealmente independientes, existen 2^k palabras código distintas, y por lo tanto, se dice que el código tiene dimensión k.

Con estas características, tendremos un código (n,k) en el que se utilizan n símbolos para codificar k símbolos de información.

4.- Linealidad. Si X e Y son dos palabras código, también lo es X + Y.

$$H(X + Y)^t = HX^t + HY^t = 0$$

Si C es un elemento de GF(p) y X es una palabra código, también lo es CX.

$$H(CX)^t = CHX^t = 0$$

Por lo tanto, el código es lineal.

3.4.3. Modelo de error.

Supongamos que la información $u = (u_1, u_2, \ldots, u_k)$ la codificamos como la palabra código $X = (X_1, X_2, \ldots, X_n)$ y se produce un error que la transforma en el vector $Y = (Y_1, Y_2, \ldots Y_n)$. Definimos el vector de error e como sigue:

$$e = Y - X = (e_1, e_2, \ldots, e_n)$$

Si únicamente perseguimos detectar si se producen errores, el decodificador debe limitarse a indicar si este vector de error es distinto de cero. Si queremos corrección de error (y el código tiene potencia para ello), el decodificacor debe, partiendo del vector erróneo Y, establecer cual es la palabra código X que tendríamos si no se hubiese producido el error. El mejor modo sería que el decodificador fuera capaz de encontrar el valor de e para hacer $X = Y + e$. Sin embargo, lo mas normal es que nunca se llegue a conocer el valor de e. La estrategia a seguir por el decodificador es entonces encontrar, partiendo de Y, la palabra código mas cercana a dicho vector. Es necesario entonces definir los conceptos de distancia y peso.

- Distancia de Hamming entre los vectores:

$X = (X_1, X_2, \ldots X_n)$ e $Y = (Y_1, Y_2, \ldots Y_n)$

es el número de componentes en que difieren. Su notación es d(X,Y).

- Peso (de Hamming) de un vector $X = (X_1, X_2, \ldots, X_n)$ es el número de componentes X_i de éste que son distintos de cero. Su notación es W(X). Es evidente que se verifica: $d(X,Y) = W(X-Y)$.

Hasta ahora caracterizábamos un código por dos parámetros, su longitud n y su dimensión k. El tercer parámetro fundamental que caracteriza un código es su distancia mínima d.

- Distancia mínima d de un código es el mínimo de las distancias entre sus palabras código.

$$X,Y \in C,\ X \neq Y,\ d = min(d(X,Y)) = min(W(X-Y)).$$

Si d es la distancia mínima de un código, cualquier par de palabras código difieren al menos en d componentes. De este modo, a un código lineal de longitud n, dimensión k y distancia mínima d se le llama código (n,k,d).

Para encontrar la distancia mínima de un código lineal no es necesario comparar dos a dos todas las palabras código.

Teorema 3.1.

La distancia mínima de un código lineal es el mínimo de los pesos de sus palabras código distintas de cero.

Demostración.

Si X e Y son palabras código, u = X-Y también lo es, así que como d(X,Y) = W(X-Y), tendremos que:

$d = min(W(u))$ c.q.d.

La aplicación de este teorema para encontrar el valor d sería util cuando tuvieramos una descripción del código por extensión. Como normalmente utilizamos la descripción mediante matrices, con el siguiente teorema (Peterson 1961) vemos como obtener la distancia mínima d a partir de la matriz de paridad H.

Teorema 3.2.

Sea C un código lineal cuya matriz de paridad es H. Para cada palabra con peso de Hamming W, existe una relación de dependencia lineal entre W columnas de H y reciprocamente, para cada relación de dependencia lineal

entre W columnas de H, existe una palabra código de peso W.

Demostración.

Un vector $X = (X_1, X_2, \ldots, X_n)$ es una palabra código si y solo si se verifica que $HX^t = 0$, o bién, denotando por h_i el i-esimo vector columna de H:

$$\sum_{i=1}^{n} h_i X_i = 0$$

Esta es la relación de dependencia lineal entre las columnas de H, y el número de columnas que aparecen con coeficientes no nulos es el número de componentes X_i no nulos de X, que es exactamente el peso de X.

Igualmente, los coeficientes de cualquier relación de dependencia lineal entre columnas de H son componentes de un vector que debe ser palabra código. c.q.d.

Corolario 3.1.

Un código lineal con matriz de paridad H tiene peso mínimo, y por tanto distancia mínima al menos d si y solo si cualquier combinación de d-1 o menos columnas de H es linealmente independiente.

3.4.4. Detección y corrección con códigos lineales.

Ahora relacionaremos la distancia mínima de un código lineal con su capacidad para detectar y corregir errores.

Teorema 3.3. (Lin 1970)

Un código lineal con distancia mínima d puede corregir $\lfloor \frac{1}{2}(d-1) \rfloor$ errores ($\lfloor X \rfloor$ = parte entera de X). Si d es par, el código puede simultaneamente corregir $\frac{1}{2}(d-2)$ errores y detectar d/2 errores.

Demostración.

Supongamos d = 3. La esfera de radio r y centro la palabra código X contiene todos los vectores Y tales que se verifique $d(X,Y) \leq r$. Si se dibuja una esfera de radio unidad alrededor de cada palabra código, estas esferas no pueden solaparse. De modo que, si tenemos la palabra código X y se produce un error que la transforma en el vector a, éste está dentro de la esfera que rodea a X, y está mas cerca de X que de ninguna otra palabra código. Así, el decodificador será capaz de corregir el error.

Igualmente, si $d = 2t+1$, las esferas de radio t alrededor de cada palabra código tampoco se solapan, y el decodificador podrá corregir t errores.

Supongamos ahora que d es par. Las esferas de radio $\frac{1}{2}(d-2)$ alrededor de las palabras código no se solapan, y el código puede corregir $\frac{1}{2}(d-2)$ errores. Si ocurren d/2 errores, el vector a queda a la misma distancia de dos o mas palabras código. En este caso el decodificador puede solamente detectar que se han producido d/2 errores. c.q.d.

Por otra parte, si ocurren mas de d/2 errores, el vector puede estar mas cerca de una palabra código incorrecta que de la correcta. Si ocurre esto, el decodificacor corregirá de un modo erróneo.

3.4.5. Decodificación de un código lineal.

Consideremos un código lineal (n, k, d) con matriz generadora G y matriz de paridad H. Sea X una palabra código. Si se produce un error, X se transforma en otro vector Y tal que $Y = X + e$. El decodificador no conoce ni X ni e. Su misión es recuperar X a partir del vector erróneo Y. Podemos caracterizar los errores mediante el síndrome.

Síndrome de un vector Y de n componentes es un vector S de $n - k$ componentes que verifica:

$$S = HY^t$$

Por la definición de código lineal ($HX^t = 0$), el síndrome será cero si Y es una palabra código, y será distinto de cero si Y no es una palabra código.

Vemos claramente que si únicamente se trata de detectar errores, basta con que el decodificador calcule el síndrome correspondiente y vea si es o no cero. El siguiente teorema nos da una propiedad importante del síndrome.

Teorema 3.4.

Dado un código lineal con matriz de paridad H, el síndrome de un vector erróneo es igual a la suma de las columnas de la matriz H correspondientes a las posiciones del vector en que se ha producido error.

Demostración.

Si se produce error en los lugares a, b, c, ..., el error será:

$$e = 0\ldots\underset{a}{1}\ldots\underset{b}{1}\ldots\underset{c}{1}\ldots 0$$

Como el síndrome es $S = HY^t = H(X+e)^t = He^t$, será:

$$S = \sum_i h_i e_i = h_a + h_b + h_c + \ldots$$

Siendo h_i la columna i de la matriz H. c.q.d.

Existe entonces una correspondencia unívoca entre cada síndrome y cada uno de los errores que el código es capaz de corregir. Hay un método general de decodificación para los códigos lineales que es el llamado de la "matriz standard", pero tiene el inconveniente de que es poco práctico y muy lento, pues necesita tener almacenados 2^{n-k} vectores y ejecutar gran número de cálculos (Peterson 1961). Existen métodos mas prácticos de deco-

dificación, pero éstos requieren fijarse en las propiedades particulares del código que estamos utilizando.

3.4.6. Códigos de Hamming.

Los códigos de Hamming forman una familia de códigos lineales que son fáciles de codificar y decodificar. Por ello han encontrado aplicación en muchas áreas. Describiremos primero los correctores de un solo error.

Como el síndrome de un vector es igual a la suma de las columnas de la matriz H correspondientes a los lugares donde se ha producido error, para obtener un código corrector de un solo error bastará con que hagamos que se verifiquen las siguientes condiciones:

1) Ninguna de las columnas de H debe ser nula. Si hubiera alguna nula, un error en la posición correspondiente a ésta daría un síndrome nulo quedando el error indetectado.

2) Todas las columnas de H deben ser distintas. Si hubiera dos columnas iguales, los errores en estas posiciones no podrían distinguirse.

Si tomamos H con r filas, es decir, que el código tiene r bits de paridad, contamos únicamente con $2^r - 1$ columnas distintas entre sí y distintas de cero ($2^r - 1$ síndromes distintos). Así pues, un código de Hamming de longitud $n = 2^r - 1$ ($r \geq 2$) tiene una matriz de paridad H cuyas columnas son todos los vectores binarios distintos de cero de longitud r, usado cada uno una sola vez. Por tanto, este código tiene $n = 2^r - 1$, $k = 2^r - 1 - r$, $d = 3$.

Ejemplo. Vamos a construir un código de Hamming corrector de un solo error para proteger vectores de información de 4 bits, y vamos a ver el proceso de codificación y decodificación.

Como tenemos $k = 4$, debe ser $r = 3$, de modo que es $n = 7$, por lo que formaremos un código (7, 4, 3). Para formar la matriz H tomamos como

columnas todos los vectores no nulos de tres bits:

$$H = \begin{matrix} 0\,0\,0\,1\,1\,1\,1 \\ 0\,1\,1\,0\,0\,1\,1 \\ 1\,0\,1\,0\,1\,0\,1 \end{matrix}$$

Reordenamos las columnas para hacer el código sistemático y queda:

$$H = \begin{matrix} 0\,1\,1\,1\,1\,0\,0 \\ 1\,0\,1\,1\,0\,1\,0 \\ 1\,1\,0\,1\,0\,0\,1 \end{matrix}$$

Esta es la matriz de paridad del código. Como es de la forma $(A \circ I)$, la matriz generadora es de la forma $(I \circ A^t)$, es decir,

$$G = \begin{matrix} 1\,0\,0\,0\,0\,1\,1 \\ 0\,1\,0\,0\,1\,0\,1 \\ 0\,0\,1\,0\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1 \end{matrix}$$

La codificación se hace del siguiente modo: Sea un vector de información $u = (u_1, u_2, u_3, u_4)$ el dato a codificar. La palabra código correspondiente es $X = uG$

$$X = (u_1, u_2, u_3, u_4, u_2+u_3+u_4, u_1+u_3+u_4, u_1+u_2+u_4)$$

que tiene una construcción física muy simple a base de puertas "O" exclusivo

El cálculo del síndrome correspondiente a una palabra código X es también muy sencillo:

$$S = HY^t = \begin{matrix} Y_2 \oplus Y_3 \oplus Y_4 \oplus Y_5 \\ Y_1 \oplus Y_3 \oplus Y_4 \oplus Y_6 \\ Y_1 \oplus Y_2 \oplus Y_4 \oplus Y_7 \end{matrix}$$

Este síndrome será el vector $\begin{bmatrix}0\\0\\0\end{bmatrix}$ si no se ha producido error, y si se ha producido un error simple, será exactamente una de las columnas de la matriz H. Identificando la posición de dicha columna tenemos localizado el error. La corrección puede hacerse mediante la inversión del bit erróneo.

A partir de un código de Hamming con distancia 3 podemos obtener fácilmente un código con distancia 4. Sea una matriz H de dimensiones $r \times (2^r-1)$ la matriz de paridad de un código de Hamming con distancia 3. Tomemos una matriz unidad de dimensión 1×2^r y una matriz 0 de dimensión $r \times 1$. Entonces H_1 es la matriz de paridad de un código de Hamming con distancia 4.

$$H_1 = \left[\begin{array}{c|c} H & 0 \\ \hline 1 & \end{array}\right]$$

Ejemplo. Partiendo de la matriz H del ejemplo anterior, obtenemos:

$$H_1 = \begin{matrix} 0&1&1&1&1&0&0&0 \\ 1&0&1&1&0&1&0&0 \\ 1&1&0&1&0&0&1&0 \\ 1&1&1&1&1&1&1&1 \end{matrix}$$

Esta construcción añade un bit redundante extra, que representa la paridad tanto de los bits de información como de los redundantes. El código resultante tiene $2^r - r - 1$ símbolos de información y $r + 1$ símbolos redundantes. Comprobamos que este código tiene distancia 4 porque ahora no hay tres columnas de H que sean linealmente dependientes.

3.4.7. Códigos b-adyacentes.

Hasta ahora hemos visto que un código es un subespacio vectorial sobre un campo finito GF(p), y hemos visto casos en los que $p = 2$. Trabajaremos ahora sobre el campo finito $GF(2^b)$.

En este caso, la corrección de un solo error va a ser equivalente a la corrección de un bloque de b bits en el campo binario. Comprobamos ade-

más que los códigos sobre $GF(2^b)$ (llamados b-adyacentes), tienen en general poca redundancia, puesto que en la mayoría de los casos, el hecho de incrementar en uno el número de símbolos redundantes, incrementa también en uno la distancia mínima del código. Estos códigos son muy útiles para proteger sistemas divididos en rodajas de b bits.

3.4.7.1. Códigos tipo Hamming sobre $GF(2^b)$

Los códigos de Hamming correctores de un solo error pueden construirse con símbolos de cualquier campo finito. Si tomamos un campo finito F, podemos construir la matriz de paridad H con símbolos de F del siguiente modo:

Tomamos como columnas de H todas las r-tuplas de elementos de F, de modo que ninguna columna dependa linealmente de otra. Esto es facil de conseguir una vez definidas en F las reglas de suma y producto. De este modo, como ninguna combinación lineal de dos o mas columnas de H es cero, el código tiene distancia mínima 3 y podrá corregir cualquier error simple.

Si utilizamos el campo GF(2), cada símbolo de $GF(2^b)$ es equivalente a una b-tupla binaria, y por ello pueden corregirse todos los errores ocurridos en bloques de b bits correspondientes a los elementos de $GF(2^b)$, siempre y cuando no se produzca error simultaneamente en mas de un bloque.

En la práctica, para poder utilizar estos códigos es necesario obtener su matriz de paridad H en forma binaria en lugar de con símbolos de $GF(2^b)$. El procedimiento para transformar una matriz de símbolos de $GF(2^b)$ en una binaria fue encontrado por Bossen (Bossen 1970) y lo vemos a continuación.

3.4.7.2. Descripción matricial.

Como $GF(2^b)$ es un espacio vectorial de dimensión b sobre el campo GF(2), la suma de elementos de $GF(2^b)$ corresponde a la suma bit a bit (módulo 2) de sus correspondientes representaciones vectoriales. La multipli-

cación puede definirse en $GF(2^b)$ como un conjunto de transformaciones lineales en el correspondiente espacio vectorial.

Sea D la representación binaria de un elemento d de $GF(2^b)$. Definimos una transformación lineal T_d como:

$T_d(\psi) = D\psi$ para todo ψ de $GF(2^b)$.

$D\psi$ es la representación vectorial del elemento $d\psi$ de $GF(2^b)$. Cada transformación lineal puede representarse por una matriz de dimensiones bxb de elementos de GF(2).

En particular, el elemento identidad (1) de $GF(2^b)$ es equivalente a la matriz identidad de dimensión bxb, asi como el elemento nulo (0) es equivalente a la matriz nula de dimensiones bxb.

Puesto que $GF(2^b)$ es equivalente al anillo de las clases de residuos GF(2)(X) mod p(X), siendo p(X) un polinomio irreducible de grado b sobre GF(2), podemos considerar el espacio vectorial correspondiente a $GF(2^b)$ descrito por los vectores:

$X^{b-1}, X^{b-2}, \ldots, X, 1$, o bien:

$$\begin{bmatrix}1\\0\\\cdot\\\cdot\\\cdot\\\cdot\\0\end{bmatrix}, \begin{bmatrix}0\\1\\\cdot\\\cdot\\\cdot\\\cdot\\0\end{bmatrix} \cdots \begin{bmatrix}0\\0\\\cdot\\\cdot\\\cdot\\1\\0\end{bmatrix} \begin{bmatrix}0\\0\\\cdot\\\cdot\\\cdot\\0\\1\end{bmatrix}$$

Así, la matriz correspondiente a la transformación lineal T_d viene dada por la concatenación de columnas:

$$T_d = (X^{b-1}d, X^{b-2}d, \ldots, Xd, d)$$

Con esta definición queda claro que la multiplicación en $GF(2^b)$ de dos elementos d_1 y d_2 es equivalente al producto ordinario vector-matriz del vector d_2 por la matriz T_{d1}. De este modo, la matriz binaria H buscada para un código dado, se construye sustituyendo cada elemento d de $GF(2^b)$ por la matriz de dimensión bxb T_d.

Ejemplo. Utilizaremos en este ejemplo símbolos de $GF(2^2)$. Tomamos $GF(2^2) = GF(2)(X) \bmod p(X)$. p(X) es el polinomio irreducible X^2+X+1. En este caso, $GF(2^2)$ consta de los elementos 0, 1, a, a^2, que corresponden a los residuos módulo p(X) de 0, 1, X y X+1. En forma vectorial estos elementos son: $\begin{matrix}0\\0\end{matrix}$, $\begin{matrix}0\\1\end{matrix}$, $\begin{matrix}1\\0\end{matrix}$, $\begin{matrix}1\\1\end{matrix}$.

Las reglas de suma y multiplicación se dan en las tablas siguientes:

+	0	1	a	a^2
0	0	1	a	a^2
1	1	0	a^2	a
a	a	a^2	0	1
a^2	a^2	a	1	0

x	0	1	a	a^2
0	0	0	0	0
1	0	1	a	a^2
a	0	a	a^2	1
a^2	0	a^2	1	a

Las matrices correspondientes de dimensión 2x2 quedan:

$$T_0 = \begin{matrix}0&0\\0&0\end{matrix} \qquad T_1 = \begin{matrix}1&0\\0&1\end{matrix} \qquad T_2 = \begin{matrix}1&1\\1&0\end{matrix} \qquad T_3 = \begin{matrix}0&1\\1&1\end{matrix}$$

En cuanto a la decodificación de estos códigos, un método posible es la utilización de un conjunto de $(2^b - 1)N$ puertas "Y", para reconocer todos los posibles síndromes. N es la longitud de la palabra código en símbolos de $GF(2^b)$.

3.5. Códigos aritméticos.

Hemos visto que los códigos de tipo paridad son útiles para proteger tanto transmisión como almacenamiento de datos en un computador. Sin

embargo, nos encontramos con el inconveniente de que estos códigos no son cerrados respecto de las operaciones aritméticas (un código es cerrado respecto de una operación "." si para cualquier A, B ϵ C se verifica que A.B ϵ C). Si usáramos como entradas a un procesador aritmético (por ejemplo un sumador) datos codificados en un código de tipo paridad, su salida en general no sería una palabra código y, por tanto, debería recodificarse. Además el código no sería de gran ayuda para la protección contra errores en el propio procesador aritmético. Podemos utilizar técnicas de predicción de paridad, pero lo mas cómodo es buscar códigos que sean cerrados respecto de las operaciones aritméticas. Este tipo de códigos es el que describimos a continuación. Primero establecemos el modelo de error aritmético.

3.5.1. Modelo de error aritmético.

3.5.1.1. Generalidades.

Los fallos o malfunciones de componentes producen como sabemos fallos lógicos, y estos pueden generar resultados erróneos o errores. Los errores en un procesador aritmético son desviaciones en sus salidas. Decimos que se produce un error en una operación de un procesador cuando el resultado obtenido R_o difiere del resultado esperado R_e definido por las especificaciones. Por tanto, definimos el error como la diferencia entre estos:

$$E = R_o - R_e$$

Para los códigos de paridad caracterizábamos un error por su peso y distancia de Hamming. Veamos con un simple ejemplo que estos conceptos de peso y distancia de Hamming no son apropiados para caracterizar los errores que se producen en los procesadores aritméticos.

Supongamos que queremos sumar los dos números binarios 000001 y 01111. El resultado correcto es evidentemente R_e = 100000. Si se produce un fallo simple que cambia el 1 del primer operando en un 0, el resultado que se obtiene es R_o = 11111, cuya distancia de Hamming al resultado esperado

es d=6. Vemos entonces que el peso de Hamming de un error aritmético puede ser mucho mayor que el número de fallos simples necesario para producirlo.

En el modelo aritmético de error que se plantéa, en lugar de ver un error como el cambio de un cierto número de bits en un vector binario, consideramos su efecto en el entero representado por dicho vector. De este modo, el efecto de un error aritmético en la representación binaria de un número entero N es transformarla en la representación binaria del entero N + E, donde E representa el "valor de error". Definimos entonces para cada vector de error un valor de error de la forma:

$$E = \sum_{i=0}^{n-1} e_i \, 2^i, \qquad y \qquad E = R_o - R_e$$

El valor absoluto del valor de error se llama magnitud del error. En el ejemplo anterior, el valor de error es -1 y la magnitud del error es 1. Es evidente que un fallo que afecta a un único bit de entrada a un sumador produce una magnitud de error que es potencia de dos, independientemente del numero de bits erróneos en el resultado. Una de las propiedades mas significativas de un error es su peso aritmético, que se define como sigue:

El peso aritmético de un número entero N es el mínimo número de términos no nulos en una expresión de la forma:

$$N = a_n r^n + \ldots + a_1 r + a_0$$

donde a_i puede ser positivo, negativo o cero, pero es menor en valor absoluto que r. Se denota por W(N) (Peterson 1961).

La distancia aritmética entre dos números enteros N_1 y N_2 es el peso de su diferencia $N_1 - N_2$. Si tenemos un número N_1 y se convierte en otro $N_2 = N_1$, y si la distancia entre N_1 y N_2 es d, se dice que ha ocurrido un error de peso d. Así, la ocurrencia de un error de peso d en un número N_1 equivale a sumarle un número de peso d. Hay que tener en cuenta que, con esta definición, la distancia depende de la base r.

Es facil demostrar que, al igual que con la distancia de Hamming, un código con distancia aritmética mínima igual o mayor que d es capaz de detectar todos los errores de peso aritmético d - 1 o menor. Para corregir cualquier combinación de t o menos errores, es necesaria y suficiente una distancia mínima d = 2t + 1, y para corregir cualquier combinación de t o menos errores, y al mismo tiempo detectar d o menos errores, se necesita una distancia mínima mayor o igual a t + d +1.

Veremos a continuación las propiedades del peso aritmético en el caso binario.

El peso aritmético binario de un número N es el mínimo número de términos no nulos en una expresión de la forma:

$N = a_n 2^n + \ldots + a_1 2 + a_0$ donde a_i=1, 0 o -1. Por ejemplo, el número decimal 31 tiene como representación binaria 11111, pero ésta no es su forma mínima. Su forma mínima es $10000\bar{1}$ ($\bar{1}$ denota -1). Así pues, W(31) = 2, que es el número de términos no nulos en su forma mínima. Un número puede tener mas de una forma mínima, por ejemplo, 25 = 011001, y 25 = $10\bar{1}001$. Existe sin embargo una forma canónica para cada número que tiene un número mínimo de términos no nulos.

Teorema 3.5. (Peterson 1961).

Para cualquier número entero N existe una representación única de la forma $N = a_n 2^n + \ldots + a_0$ en la que a_i= +1, -1 o 0, y en la que no existen dos a_i consecutivos no nulos. Esta representación además tiene el mínimo número de términos distintos de cero.

Demostración.

Demostraremos primero la existencia de una representación de este tipo con mínimo número de términos no nulos, y la unicidad de la misma.

Sea $N = b_n 2^n + \ldots + b_0$ (2)

Si existen mas de dos b_i consecutivos no nulos, tomemos el par con i menor, $b_{i+1}2^{i+1} + b_i 2^i$. Si se verifica $b_i = -b_{i+1}$, entonces $b_{i+1}2^{i+1} + b_i 2^i = b_{i+1}2^i$, y la expresión resultante tiene menos unos. Si es $b_i = b_{i+1}$, entonces $b_{i+1}2^{i+1} + b_i 2^i = b_{i+1}2^{i+2} - b_i 2^i$. Si hacemos la sustitución en la ecuación (2), el coeficiente de 2^{i+1} se hace cero. El término $b_{i+1}2^{i+2}$ puede combinarse con el término existente $b_{i+2}2^{i+2}$. Si b_{i+2} es cero, la nueva expresión tiene el mismo número de términos no nulos que antes. Si $b_{i+2} = -b_{i+1}$, estos términos se cancelan y la expresión resultante tiene menos términos. Si $b_{i+2} = b_{i+1}$ los combinamos resultando $b_{i+2}2^{i+2} + b_{i+1}2^{i+2} = b_{i+1}2^{i+3}$, y el coeficiente de 2^{i+2} se hace cero, mientras que $b_{i+1}2^{i+3}$ puede combinarse con $b_{i+3}2^{i+3}$. El "acarreo" puede propagarse hasta el término de mayor orden, pero cada vez que se produce un acarreo, el número de unos disminuye. De modo que la expresión resultante tiene la misma forma general, no tiene mas términos no nulos que la de partida, y además no existen dos términos sucesivos distintos de cero de grado menor que i+2.

Repitiendo el proceso, resultará una expresión que no tiene dos términos consecutivos no nulos, ni mas términos no nulos que la expresión de partida. El proceso tendrá fin puesto que partimos de un número finito de términos.

Si la expresión original tenía el número mínimo de términos no nulos, éste no puede decrementarse, y la expresión final tendrá el mismo número de términos no nulos. Puesto que para cada número existe al menos una expresión de la forma (2) con número mínimo de términos, se sigue que para cualquier número existe una expresión del mismo tipo con número mínimo de términos no nulos y ningún par de términos consecutivos no nulos.

Queda por demostrar que la expresión es única. Veamos primero que en una expresión del tipo (2) con todos los coeficientes +1, -1 o 0, si el primer coeficiente es +1, el menor valor de N aparece cuando el resto de los coeficientes valen -1, y entonces:

$$N_{min} = 2^n - 2^{n-1} - \ldots - 1 = 1$$

Del mismo modo, si el primer coeficiente es -1, el mayor valor de N posible es -1. Así, una expresión con $a_n \neq 0$ puede ser cero si y solo si todos sus coeficientes son cero. Ahora definimos:

$$N_1 = a_n 2^n + \ldots + a_1 2 + a_0$$

$$N_2 = b_n 2^n + \ldots + b_1 2 + b_0$$

y suponemos que en ninguna de las dos expresiones hay dos coeficientes consecutivos no nulos, y que para al menos un i es $a_i \neq b_i$. Entonces:

$$N_2 - N_1 = (b_n - a_n)2^n + \ldots + (b_1 - a_1)2 + (b_0 - a_0)$$

Si $a_i = b_i$ hay dos casos: 0 bien $a_i = -b_i$ o bien uno de los dos términos es cero. En el último caso, la expresión $N_2 - N_1$ tiene un término distinto de cero, puesto que el acarreo a la posición i+1 es imposible. En el primer caso resulta el término $2b_i 2^i = b_i 2^{i+1}$. Puesto que ni N_1 ni N_2 tienen dos términos consecutivos distintos de cero, $b_{i+1} = a_{i+1} = 0$, y el término $b_i 2^{i+1}$ permanece en la expresión de $N_2 - N_1$. Si todos los pares $a_i 2^i$, $b_i 2^i$ se combinan de este modo, la expresión resultante tiene la forma (2). No todos los coeficientes son nulos, y por lo tanto es $N_2 - N_1 \neq 0$. Así pues, dos expresiones de este tipo distintas representan a números distintos. c.q.d.

Esta demostración nos muestra un método para expresar un número en su forma canónica (recodificación canónica) y de este modo determinar su peso aritmético.

Otra propiedad importante del peso aritmético es que:

$$W(N) = W(-N)$$

Esto puede comprobarse de un modo trivial en la expresión (1) donde cada a_i puede ser positivo, negativo o cero. Si se verifica que

$N = \sum_{i=0}^{n} a_i r^i$, entonces es

$$-N = \sum_{i=0}^{n} (-a_i) r^i,$$

y ambos tienen exactamente el mismo número de términos no nulos en sus formas canónicas.

El peso aritmético cumple también la desigualdad triangular:

$$W(N_1 + N_2) \leq W(N_1) + W(N_2)$$

Esto está claro si se considera la suma de dos números N_1 y N_2 que están expresados en forma canónica. En su suma pueden producirse acarreos o cancelación de términos no nulos, pero el número de éstos en la suma no puede exceder a la suma de los términos no nulos de N_1 y N_2.

Si cada fallo simple en un sumador produce un valor de error de $+2^i$ para algun i, entonces se necesitan al menos k fallos simples para producir un error de peso k. Inversamente, un fallo múltiple en k componentes produce un peso de error como máximo de k. Un fallo simple en un sumador produce un valor de error de $\pm 2^i$ si los bits de suma y acarreo se calculan mediante circuitos independientes. Sin embargo, los fallos simples en sumadores con otra estructura pueden producir valores de error que no son potencias de dos.

3.5.1.2. Errores en un conjunto finito.

Puesto que en un computador los números enteros se representan mediante un número finito de bits, existe un máximo número representable m para cada arquitectura particular. Al sumar dos enteros lo que ejecutamos en definitiva es su suma módulo m. Para cualquier par de enteros N_1 y N_2, la suma calculada es $(N_1 + N_2)$ módulo m. En representación en complemento a 2 con n bits, es $m = 2^n$, y en representación de complemento a 1 es $m = 2^n - 1$.

De este modo los operandos N_1 y N_2 están limitados a un conjunto finito de valores. Sea Z_m el anillo finito de los enteros módulo m, es decir, (0,1,...,m-1). El inverso aditivo de N en Z_m se llama complemento de N y se denota por $\bar{N} = m - N$, siendo

$$\bar{N} = -N \text{ (módulo m)}$$

El peso aritmético como lo definíamos en el apartado anterior satisfacía la igualdad W(N) = W(-N). Sin embargo, para N y $\bar{N}$ en el anillo Z_m, W(N) puede no ser igual a $W(\bar{N})$. Por ejemplo, consideremos Z_{63}, donde $\overline{32} = 63 - 32 = 31$. Tenemos:

$$W(32) = 1; \quad W(\overline{32}) = W(31) = 2$$

Esto es algo indeseable puesto que en este caso la magnitud de error no especifica unívocamente su peso aritmético. Por ejemplo, un inversor cortocircuitado en un sumador puede generar errores de valor $+2^j$ y -2^j en diferentes ocasiones, y por tanto, provocar errores de diferentes pesos. Para paliar esta dificultad Rao (Rao 1968) introdujeron el concepto de peso aritmético modular para números en un anillo finito.

El peso modular de un número entero N de Z_m, denotado por $W_m(N)$ viene dado por la expresión:

$$W_m(N) = \min(W(N), W(\bar{N}))$$

La distancia modular entre dos números N_1 y N_2 de Z_m, denotada por $d_m(N_1,N_2)$ viene dada por:

$$d_m(N_1,N_2) = W_m(N_1-N_2) = W_m(N_2-N_1)$$

En general para números en Z_m no se verifica la desigualdad triangular:

$$W_m(|N_1+N_2|_m) \leq W_m(N_1) + W_m(N_2)$$

Veamos con un ejemplo el hecho de que debemos considerar el máximo número representable m en la definición de peso aritmético. Consideremos un error que cambia la representación en 4 bits de 0 (0000) en 15 (1111). El valor de error es aparentemente 15, y el peso del error es 4. Pero si la suma se está haciendo en aritmética de complemento a 1, es m = 15 y además 0000 y 1111 representan el mismo número, luego no se ha producido error. Si estuvieramos usando aritmética de complemento a 2, sería m = 16, y el valor del error sería -1 (mod. 16). El peso del error sería 1.

3.5.1.3. Conjuntos de error.

Consideremos R_e, R_o y E enteros pertenecientes a Z_m, y procedamos a definir las clases de error en Z_m. Denotamos el conjunto de todos los errores en Z_m con peso modular igual a d por V(m,d), y el conjunto de todos los errores en Z_m de peso modular menor o igual que d por U(m,d). El conjunto de todos los errores de peso modular 1 se llama conjunto de errores simples, los de peso modular 2, errores dobles, etc. También U(m,d) es igual a la unión de los conjuntos V(m,1), V(m,2), ..., V(m,d).

3.5.2. Tipos de códigos aritméticos.

Podemos clasificar los códigos aritméticos en dos tipos: Separados y no separados.

En un código no separado un dato N_1 se codifica como $N_1' = f(N_1)$. En estos códigos, la codificación de la suma de dos datos N_1 y N_2 se obtiene ejecutando una operación binaria sobre N_1' y N_2'.Es decir que:

$$f(|N_1 + N_2|_m) = N_1' * N_2'$$

El concepto se ilustra en la figura 3.1.

En un código separado, la codificación de un dato N_1 se obtiene mediante la concatenación de N_1 y un símbolo redundante, calculado a partir de N_1 mediante la aplicación de una función C. De este modo,

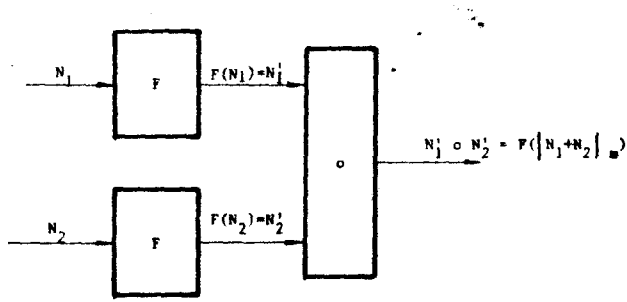


Figura 3.1

$f(N_1) = C(N_1), N_1$. Así son independientes las operaciones efectuadas sobre la parte de datos de las efectuadas sobre la parte redundante.

$$f(N_1 + N_2) = C(N_1) \circ C(N_2),\ |N_1 + N_2|_m$$

El concepto se ilustra en la figura 3.2.

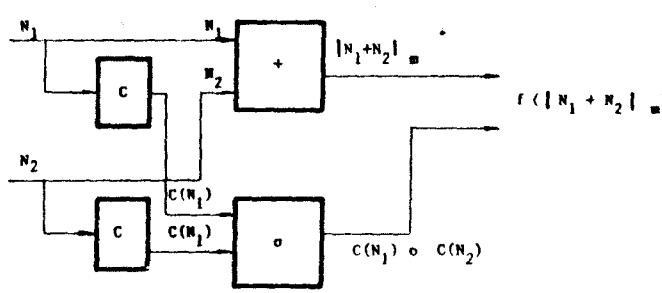


Figura 3.2

Los códigos aritméticos pueden también clasificarse como sistemáti-

cos y no sistemáticos. En un código sistemático, la codificación de un dato N_1 tiene un subcampo que es igual a N_1, y el resto está compuesto por la redundancia. Un código no sistemático no cuenta con esta propiedad.

Los códigos separados son trivialmente sistemáticos, mientras que la mayoría de los no separados son no sistemáticos.

Existen fundamentalmente tres tipos de códigos aritméticos: Códigos AN, códigos gAN y códigos de residuos.

3.5.2.1. Códigos AN.

En estos códigos, también llamados códigos de producto, la forma codificada de un número N es la representación en base r con n dígitos del producto AN, donde A es una constante llamada "generador del código".

El número de dígitos que son necesarios para representar un número N en base r, es el menor entero mayor que $\log_r N$. El número de dígitos necesario para representar AN es el menor entero mayor o igual que $\log_r AN = \log_r N + \log_r A$. La cantidad $\log_r A$ se llama redundancia del código. El número de dígitos redundantes necesarios para esta representación difiere de $\log_r A$ en menos de 1.

Consideremos la suma de dos números enteros N_1 y N_2. La suma de sus formas codificacas $AN_1 + AN_2$ es igual a $A(N_1 + N_2)$, que es la forma codificada de su suma. El resultado de la suma de dos palabras código es entonces otra palabra código siempre que $N_1 + N_2 < m$. En general se verifica que:

$$A|N_1 + N_2|_m = |AN_1 + AN_2|_{Am}$$

Así pues se puede comprobar si se han producido o no errores en la suma. Un error E en ésta provocará:

$$S = A(N_1 + N_2) + E = AN_3 + E$$

La comprobación puede consistir en verificar si la suma obtenida es una palabra código, cosa que haremos dividiendo S por A y viendo si el resto es cero. Esto es lo mismo que obtener el residuo módulo A de la suma.

$$|S|_A = |AN_3 + E|_A = |AN_3|_A + |E|_A = |E|_A$$

Si no se ha producido error es E = 0 y por tanto $|E|_A = 0$. Si E es un múltiplo exacto de A también es $|E|_A = 0$ y el error permanece indetectado. Sin embargo, se suele elegir A de modo que la probabilidad de que aparezca un error indetectable sea muy pequeña. El concepto de la detección de errores se muestra en la figura 3.3.

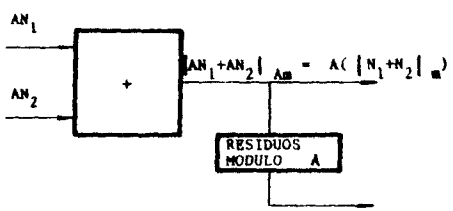


Figura 3.3

Por ejemplo, para detectar errores simples necesitamos que la distancia mínima entre palabras código sea d = 2, es decir, que no haya dos palabras del código con distancia 1. De este modo, para todos los N_1 y N_2 se verifica:

$$AN_1 - AN_2 = A(N_1 - N_2) \neq br^j, \text{ con } 0 < b < r$$

Esto puede asegurarse eligiendo A primo respecto de r y mayor que r. La elección A = r + 1 es siempre válida. Un código 3N es capaz de detectar cualquier error simple en una representación binaria. Un código 11N lo hará en una representación decimal. Así pues, la detección de errores simples necesita una cantidad fija de redundancia para una base dada, indepen-

dientemente del tamaño de los números N. La redundancia necesaria es $\log_r(r + 1)$, con lo cual no necesitamos ni menos de uno ni mas de dos dígitos extra. Esto es análogo al hecho de que con un solo símbolo de paridad se pueden detectar todos los errores simples en un código de tipo paridad, independientemente del tamaño de las palabras.

3.5.2.1.1. Corrección de errores con códigos AN.

Para un entero X denotamos por S(X) el síndrome de X tal que se verifica:

$$S(X) = |X|_A$$

Suponemos que las operaciones aritméticas de interés son cerradas respecto del código. De este modo, el resultado esperado de la operación, R_e es una palabra código, y por tanto, es múltiplo de A. Entonces, el síndrome del resultado obtenido es:

$$S(R_0) = |R_0|_A = ||R_e + E|_M|_A = ||AN + E|_M|_A \qquad (M = Am)$$

Como M es múltiplo de A, $||AN + E|_M|_A = |AN + E|_A$, y por tanto:

$$S(R_0) = |E|_A = S(E)$$

Deducimos entonces que el síndrome del resultado de una operación aritmética es el síndrome del error E. Un error $E \neq 0$ se dice que es detectable bajo A si y solo si es $S(E) = |E|_A \neq 0$.

Lema 3.1.

Un código AN en Z_M puede detectar cualquier error del conjunto U(M,d) si y sólo si para cualquier $E \in U(M,d)$ es $S(E) \neq 0$, es decir, que el conjunto de errores U(M,d) es detectable bajo A (generador del código) si y sólo si el código AN es capaz de detectar todos los errores de peso modular d o menor.

Además, si dos errores no nulos y distintos E_1 y E_2 tienen sus síndromes no nulos y distintos, dichos errores se llaman distinguibles bajo A. Si cualquier par distinto de errores E_i y E_j U(M,t) es distinguible bajo A, los síndromes de todos los errores en U(M,t) son distintos. De este modo podemos asociar cada error a un síndrome, y por lo tanto localizarlo y corregirlo.

Lema 3.2.

Un código AN puede corregir cualquier error en U(M,t) si cualquier par distinto de errores de U(M,t) es distinguible bajo A. En otras palabras, que para poder efectuar la corrección de errores debe existir una correspondencia biunívoca entre el conjunto de los errores que queremos corregir y el conjunto de todos los síndromes.

Veamos ahora la relación existente entre la distancia mínima de un código AN y su capacidad de detección y corrección de errores.

Teorema 3.6. (Rao 1974).

Un código AN en Z_M puede detectar cualquier error $E \in U(M,d)$ si y solo si la distancia modular del código es al menos d + 1. También un código AN en Z_M puede corregir cualquier error $E \in U(M,t)$ si y solo si la distancia modular del código es al menos 2t + 1.

De un modo mas general, un código AN puede detectar cualquier error en U(M,d) y corregir cualquier error en U(M,t) si y solo si la distancia modular del código es al menos t + d + 1.

Los códigos con distancia mínima mayor que 2 pueden describirse en términos de un número $M_r(A,d)$ que se define como sigue (Peterson 1961):

$M_r(A,d)$ es el menor entero positivo tal que su producto por A tiene peso aritmético menor que d. Es decir:

$$W_M(AM_r(A,d)) < d$$

En el caso binario es r = 2 y denotamos este entero por $M_2(A,d)$ o bien únicamente por M(A,d).

Teorema 3.7.

Para todo A y r, si N se restringe al rango $0 \leqslant N \leqslant M_r(A,d)$ o al rango $-\frac{1}{2} M_r(A,d) \leqslant N < \frac{1}{2} M_r(A,d)$, el código AN tiene distancia mínima d.

Demostración.

Si N_1 y N_2 están ambos dentro del rango, se verifica que $N_1 - N_2 < M_r(A,d)$ y $AN_1 - AN_2 < AM_r(A,d)$ y por lo tanto, por definición de $M_r(A,d)$, $AN_1 - AN_2$ tiene peso mayor que d - 1.

El problema ahora es encontrar el rango de información m para un entero impar dado A, para lograr corrección de errores simples (distancia 3).

La longitud de r módulo A, para A y r primos entre sí, es el menor entero positivo i tal que $r^i = \pm 1$ (mod. A). Llamamos a este entero $L_r(A)$. Si la base es r = 2, lo llamamos simplemente L(A).

El orden de r módulo A, para A y r primos entre sí es el menor entero positivo i tal que $r^i = 1$ (mod. A). Lo denotamos por $e_r(A)$. Para r = 2 lo denotamos simplemente por e(A). También se le conoce por el nombre de "exponente".

Ejemplos de longitud y orden son los siguientes: $L_2(11) = 5$, puesto que $2^5 = -1$ (mod. 11); pero su orden es $e_2(11) = 10$, puesto que el menor entero positivo i que verifica $2^i = 1$ (mod. 11) es 10. Por otro lado, la longitud de 2 módulo 7 y el orden de 2 módulo 7 tienen el mismo valor, L(7) = e(7) = 3

En el anillo Z_A, las potencias sucesivas de r (para A y r primos entre sí) generan un subgrupo multiplicativo en Z_A. El conjunto de los enteros en Z_A que son primos respecto de A forman un grupo multiplicativo G(A), llamado sistema de residuos reducido. El grupo multiplicativo generado por r en Z_A es un subgrupo de G(A). En este caso G(A) está formado por todos los elementos no nulos de GF(A).

Si r genera completamente el grupo G(A), entonces a r se le llama elemento primitivo en Z_A. En GF(A), las potencias de r completan todos los elementos no nulos de GF(A) si y solo si r es un elemento primitivo de GF(A).

Consideremos por ejemplo GF(7) = Z_7. Las potencias de 2 generan el conjunto (2, 4, 8=1), y no completan todos los elementos no nulos; es $e_2(7) = 3$. Por otra parte, con las potencias de 3 se generan (3, 3^2=2, 3^3=6, 3^4=4, 3^5=5, 3^6=1), completando GF(7), y es $e_3(7) = 6$. Así, 3 es un elemento primitivo de GF(7). Para cualquier elemento primitivo r de GF(A) se verifica que $e_r(A) = A - 1$.

Teorema 3.8. (Brown).

Dado un entero positivo $A \geq 3$, $M_2(A,3)$ es el entero que para el menor k positivo satisface:

$$M_2(A,3) = (2^k \pm 1)/A \qquad (3)$$

Este teorema se obtiene naturalmente de la definición de $M_r(A,d)$. Puesto que A es impar, AN no puede ser igual a 2^k. $M_2(A,3)$ es el menor entero N tal que el peso aritmético de AN es 2 o menor. Así pues, $AM_2(A,3)$ debe tener peso 2, o en otras palabras, debe tener la forma $2^i \pm 2^j$ para $i > j$. Factorizando obtenemos que $2^i \pm 2^j = (2^{i-j} \pm 1)\, 2^j$. Puesto que 2^j es primo respecto de A, 2^j debe ser submultiplo de $M_2(A,3)$, y para algún $N < M_2(A,3)$, AN es de la forma $2^k \pm 1$. Esto contradice la definición de $M_2(A,3)$. De este modo, $2^j = 1$ y $AM_2(A,3)$ debe ser de la forma $2^k \pm 1$. c.q.d.

Si se conoce L(A), entonces M(A,3) se obtiene directamente de (3). La cuestión entonces es como encontrar L(A)

Lema 3.3.

Dados dos enteros r y A primos entre si, $L_r(A)$ debe ser o bien $e_r(A)$ o bien $e_r(A)/2$.

Demostración.

De la definición de $L_r(A)$ y $e_r(a)$ sabemos que es $L_r(A) \leq e_r(A)$. También que se verifica:

$r^{L_r(A)} \equiv 1 \pmod{A}$ si y solo si $L_r(A) = e_r(A)$.

Supongamos que es $r^{L_r(A)} \equiv -1 \pmod{A}$. Por lo tanto, es $e_r(A) \neq L_r(A)$ y se verifica que:

$$r^{2L_r(A)} \equiv 1 \pmod{A}.$$

Esto significa que $e_r(A)$ es submultiplo de $L_r(A)$. Si es $e_r(A) = 2L_r(A)$, entonces $L_r(A) = e_r(A)/2$. Si $e_r(A)$ es un divisor propio de $2L_r(A)$, entonces es $e_r(A) \leq L_r(A)$, lo que contradice los resultados anteriores. c.q.d.

Teorema 3.9.

Para un entero impar A se verifica que:

$AM_r(A,3) = 2^{L(A)} + 1$ si y solo si $2^{L(A)} \equiv -1 \pmod{A}$

$AM(A,3) = 2^{L(A)} - 1$ si y solo si $2^{L(A)} = 2^{e(A)} \equiv 1 \pmod{A}$.

Lema 3.4.

r es un elemento primitivo de un campo finito GF(A) si y solo si $e_r(A) = A - 1$ y $L_r(A) = (A - 1)/2$.

Demostración.

Consideremos GF(A). Si r es elemento primitivo, entonces $e_r(A) = A - 1$. Del lema 3.3, $L_r(A) = e_r(A)/2$ o $e_r(A)$. Si es $L_r(A) = e_r(A)/2$, entonces es $L_r(A) = (A - 1)/2$. Si $L_r(A) = e_r(A)$, entonces no existe ningun j tal que $r^j \equiv -1 \pmod{A}$. Así, r no genera -1, y por tanto no es elemento primitivo. Así, debe ser $L_r(A) = (A - 1)/2$.

Inversamente, sea $e_r(A) = A - 1$. Entonces las potencias sucesivas de r generan $S = (r, r^2, \ldots, r^{A-1}=1)$. Los elementos de este conjunto S deben ser distintos si $r^j \equiv r^i \pmod{A}$ para $i,j \leq A - 1$. Entonces, r^{j-i} o $r^{i-j} \equiv 1 \pmod{A}$, lo cual es absurdo. Además, r y A son primos entre si, igual que r^j y A para todo j. Así el conjunto S tiene A - 1 elementos distintos, cada uno primo respecto de A. Esto puede ser cierto unicamente si A es primo y si r es elemento primitivo de GF(A). c.q.d.

Teorema 3.10. (Brown, Peterson).

Siendo A un número impar primo, 2 es elemento primitivo de GF(A) si y solo si se verifica:

$$AM_2(A,3) = 2^{(A-1)/2} + 1$$

-2 pero no 2 es elemento primitivo de GF(A) si y solo si se verifica:

$$AM_2(A,3) = 2^{(A-1)/2} - 1.$$

Estos códigos cuyos generadores A son primos y tienen a 2 o -2 como elementos primitivos de GF(A) se llaman códigos de Brown Peterson. Son ca-

paces de corregir errores simples.

Un código se llama perfecto si cualquier elemento de G(A) se utiliza para corregir errores en el conjunto U(M,d) para algún d ⩾ 1. Los códigos de Brown-Peterson son perfectos.

En la tabla 3.1 se listan algunos códigos de este tipo.

A	L(A)	e(A)	M(A,3)	AM(A,3)
11	5	10	3	$2^5 + 1$
13	6	12	5	$2^6 + 1$
19	9	18	27	$2^9 + 1$
23	11	11	89	$2^{11} - 1$
29	14	28	565	$2^{14} - 1$
37	18	36	7805	$2^{18} - 1$
47	23	23	178481	$2^{23} - 1$
53	26	52	1266205	$2^{26} - 1$

Tabla 3.1.

3.5.2.2. Códigos sistemáticos no separados.

Los códigos sistemáticos pueden ser de dos tipos, separados y no separados. Como códigos separados estudiaremos los códigos de residuos. Estudiamos a continuación los códigos sistemáticos no separados haciendo énfasis en el mas práctico, que es el código gAN.

Puede observarse que dado un código AN con $2^{a-1} < A < 2^a$, podemos obtener de el un subcódigo sistemático si codificamos la información N en la forma siguiente (Peterson 1961), (Massey 1964):

$$X = N2^a + C_1(N), \text{ siendo } C_1(N) = |-N2^a|_A$$

De la expresión anterior obtenemos dos conclusiones:

-X es múltiplo de A, por tanto es una palabra código.

-Los dígitos de mayor orden de X representan (en forma binaria) la información N que queríamos codificar, y los de menor orden representan la parte redundante. Por tanto, este código es sistemático.

Tomemos como ejemplo el código 3N. Podemos obtener un subcódigo sistemático del mismo como:

$$X = N2^2 + C_1(N), \text{ donde } C_1(N) = |-N2^2|_3 = |-N|_3 = |2N|_3$$

De este modo con este código representamos un número N en la forma: $X = 4N + |2N|_3$.

Cada una de estas palabras código pertenece al código de partida, pero no todas las palabras código del AN de partida están presentes en el subcódigo sistemático.

Concluimos entonces que el código no puede ser cerrado respecto de la suma ordinaria. Este hecho queda igualmente claro cuando sumamos las palabras código que representan a dos números N_1 y N_2 usando un sumador binario convencional. El acarreo de la parte redundante se propaga a la parte de información, con lo que el resultado no siempre será la palabra código que representa a $N_1 + N_2$.

Para eliminar este inconveniente, Garner (Garner 1966) introdujo una variedad de subcódigo sistemático con los dígitos de información en la parte menos significativa de la palabra código y los dígitos redundantes en la parte mas significativa. Este tipo de códigos los estudió Rao (Rao 1972) con el nombre de códigos $|gAN|_M$.

3.5.2.2.1. Construcción de los códigos gAN.

Consideremos un código AN que representa información en el rango $0 \leq N < m$, de modo que A y m sean primos entre sí (m es de la forma 2^k o

2^{k-1}). Entonces, existe un subcódigo sistemático $|gAN|_M$ en el cual $M = Am$ y se verifica:

$$gA = C_1 m + 1 \text{ para } 0 < C_1 < A$$

gA es la palabra código que representa la unidad, y también puede denotarse por el par $(C_1,1)$. De lo anterior vemos que $C_1 m + 1$ es múltiplo de A, por lo tanto, $|C_1 m + 1|_A = 0$, de donde obtenemos que $C_1 = |-m^{-1}|_A$.

De este modo, la palabra código que representa a un número $N \in Z_m$ será:

$$|gAN|_M = |C_1 Nm + N|_M = |C_1 N|_A m + N$$

Si denotamos $C_N = |C_1 N|_A = |-Nm^{-1}|_A$, tendremos:

$|gAN|_M = C_N m + N$, que también puede representarse por (C_N,N).

De aquí obtenemos la representación binaria de la palabra código $X = |gAN|_M$ como un vector:

$$X = X_{n-1}, X_{n-2}, \ldots, X_k, X_{k-1}, \ldots, X_0 \text{ donde}$$

$$N = \sum_{i=0}^{k-1} X_i 2^i, \text{ y}$$

$$C_n = \sum_{j=k}^{n-1} X_j 2^{j-k}$$

Podemos también escribir la palabra X como:

$$X = |gAN|_M = \sum_{j=0}^{n-1} X_j V_j, \text{ donde } V_j \text{ viene dado por:}$$

$$V_j = 2^j \text{ para } j = 0,1,\ldots,k-1, \text{ y}$$

$$V_j = m\, 2^{j-k} \text{ para } j = k,k+1,\ldots,n-1$$

Veamos ahora como queda la estructura de suma para este tipo de códigos.

Consideremos las dos palabra código siguientes:

$$X = C_{N1}m + N_1, \quad Y = C_{N2}m + N_2$$

que representan respectivamente a los números N_1 y N_2 de Z_m. Su suma es:

$Z = |X + Y|_M = |(C_{N1} + C_{N2})m + N_1 + N_2|_M$. Por lo tanto,

$Z = |(C_{N1} + C_{N2} + T)m + |N_1 + N_2|_m|_M$, o bien:

$Z = |C_{N1} + C_{N2} + T)|_A m + |N_1 + N_2|_m$, don de T es:

$T = 1$ si $N_1 + N_2 \geq m$, y

$T = 0$ si $N_1 + N_2 < m$

Llamando $N_3 = |N_1 + N_2|_m$ a la suma módulo m de las partes de información de los operandos nos queda,

$$Z = |C_{N1} + C_{N2} + T|_A m + N_3$$

De aquí se sigue que Z es la palabra código que representa a N_3, con

$$C_{N3} = |C_{N1} + C_{N2} + T|_A = |-N_3 m^{-1}|_A$$

De este modo hemos obtenido la estructura de la suma de palabras código en un subcódigo sistemático. En la figura 3.4 se supone que las partes de información (k bits de orden inferior) usan un sumador módulo m, y propagan un acarreo T a la parte redundante si es $N_1 + N_2 \geq m$. Las partes redundantes C_{N1} y C_{N2} así como el acarreo T usan un sumador módulo A para obtener C_{N3}.

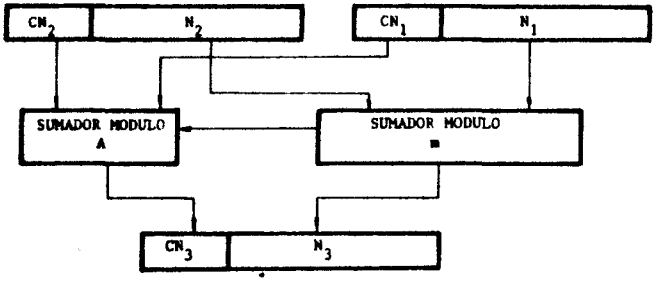


Figura 3.4

3.5.2.2.2. Corrección de errores simples.

Un error simple E en el sumador módulo M (M = Am) se define del siguiente modo:

$$E = \pm V_j, \text{ donde } V_j = 2^j \text{ para } j = 0,1,\ldots,k-1$$

$$V_j = m2^{j-k} \text{ para } j = k,k+1,\ldots,n-1$$

Un error negativo $-V_j$ equivale a $M - V_j$. Si tomamos valores arbitrarios de A y m estos errores negativos no siempre serán de la forma $+2^j$. Sin embargo, si tomamos $m = 2^k$, el conjunto de los errores simples coincide con V(M,1), que es el conjunto de errores modulares simples definido en el modelo de error aritmético. Para el caso general de A y m cualesquiera, debemos ampliar el modelo de error, definiendo el peso y la distancia sistemáticos.

El peso sistemático de un número N de Z_m, denotado por $\xi(N)$ es el mínimo número de términos no nulos en la representación de N o de $M - N = \bar{N}$ en la forma:

$$a_1 V_{j1} + a_2 V_{j2} + \cdots$$

Donde a_i es 1, 0 o -1 y V_{ji} son los valores de los dígitos definidos anteriormente. Cuando $m = 2^k$, entonces es $V_j = 2^j$ para $j = 0,\ldots,n-1$, y $\xi(N) = \xi(\bar{N}) = \min(W(N),W(\bar{N}))$, de modo que el peso sistemático coincide con el peso modular W_M definido en el modelo de error.

Para $x,y \in Zm$, sea x-y la diferencia en Zm. Entonces,

La distancia sistemática entre x e y, denotada por D(x,y) viene dada por el peso sistemático de la diferencia $\xi(x-y)$. Esta distancia cumple las siguientes propiedades (Rao 1972):

1) $D(x,y) = D(y,x)$

2) $D(x,y) \geq 0$ (la igualdad solo si $x = y$)

3) Para x,y,z de Z_m, con $m = 2^k$ y A de forma tal que la suma módulo A no necesite mas de dos realimentaciones de acarreo, entonces:

$$D(x,y) + D(y,z) \geq D(x,z).$$

Así pues, si m y A son de forma tal que se cumplen estas tres propiedades, la distancia sistemática es una métrica.

Un error E de Z_M se dice que es un error de peso d si y solo si $\xi(E) = d$.

Vemos a continuación unos teoremas que relacionan la distancia sistemática con las propiedades de control de error del código. La demostración de los mismos puede encontrarse en (Rao 1974).

Teorema 3.11.

Un código gAN tiene distancia sistemática mínima d dada por:

$m = 2^k \leq M(A,d)$

Teorema 3.12.

Supongamos que la desigualdad triangular se verifica en Z_M con la distancia sistemática. Entonces el código gAN puede detectar cualquier error E de peso d o menor si y solo si la distancia sistemática mínima del código es al menos d+1. También puede corregir este código cualquier error E de peso t o menor si y solo si la distancia sistemática mínima es al menos 2t+1.

Estos dos teoremas pueden combinarse. Las propiedades de control de error de un código AN pueden transferirse a su correspondiente subcódigo gAN con tal que la suma módulo A no necesite mas de dos realimentaciones de acarreo, y que $m = 2^k = M(A,d)$.

Por otra parte, si la suma módulo A necesita mas de dos realimentaciones de acarreo, no se verifica la desigualdad triangular y no es facil establecer las propiedades de control de error del código. Este caso fue estudiado por Varanasi y Rao (Varanasi 1972). El teorema que se enuncia a continuación resume los resultados de dicho estudio.

Teorema 3.13.

Un código gAN con $m = 2^k$ es capaz de corregir t errores si y solo si se verifica:

$m < M(A,3)$ para $t = 1$

$m < M(A,3)/2$ para $t \geq 1$.

Si queremos construir un código gAN que sea capaz de corregir todos los errores simples (de peso sistemático 1 y del tipo 2^j o $M - 2^j$), entonces A debe ser un número primo que tenga 2 ó -2 como elemento primitivo del campo de los enteros módulo A.

Por ejemplo, si A = 29, el sumador para la parte redundante tendrá cinco dígitos y necesitará dos realimentaciones de acarreo, pues 32 = 2 + 1 (mod. 29). Si A = 53, éste tendrá seis dígitos, y el número de realimentaciones de acarreo necesarias será 3, pues 64 = 8+2+1 (mod. 53). Para que se cumplan los teoremas anteriores debemos elegir A de modo que no se necesiten mas de dos realimentaciones de acarreo. En la tabla 3.2 se muestran algunos valores de A que satisfacen esta condición, y permiten la corrección de errores simples para $k \ll k_{max}$ ($2^k \ll M_2(A,3)$). k representa el número de dígitos de información y c el número de dígitos redundantes.

A	$AM_2(A,3)$	k_{max}	c
19	$2^9 + 1$	4	5
29	$2^{14} + 1$	9	5
47	$2^{23} - 1$	17	6
61	$2^{30} + 1$	24	6
121	$2^{55} + 1$	48	7

Tabla 3.2.

3.5.2.3. Códigos separados.

Consideraremos ahora las propiedades de detección y corrección de error de los códigos separados, es decir, aquellos que operan de un modo independiente con la parte de datos y con la parte redundante.

Un código separado tiene en general la forma (C(N),N), siendo C(N) el símbolo redundante correspondiente al número N. Peterson (Peterson 1961) demostró que todos los códigos separados son equivalentes a códigos de re-

siduos. Consideró la estructura de suma de la figura 3.5

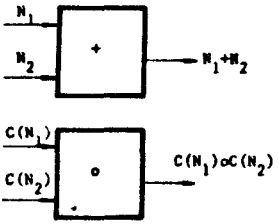


Figura 3.5

Es decir, que la suma de las dos palabras código $(C(N_1),N_1)$ y $(C(N_2),N_2)$ la definimos como:

$$(C(N_1),\ N_1) + (C(N_2),\ N_2) = (C(N_1) \circ C(N_2),\ N_1 + N_2)$$

La suma de dos palabras código implica dos operaciones + y $\circ$, en las partes de dato y redundante respectivamente, por lo tanto, para que el código sea cerrado respecto de la suma debe cumplirse:

$$C(N_1) \circ C(N_2) = C(N_1 + N_2)$$

De este modo, la detección de errores se efectua simplemente comprobando si la suma de dos palabras código es otra palabra código.

Tomando N_1 y N_2 del conjunto infinito de los números enteros, y suponiendo que el número de símbolos redundantes es menor que el número de símbolos de información (C(N) con un número finito de dígitos), Peterson obtuvo el siguiente resultado:

Teorema 3.14.

Si hay mas símbolos de información que redundantes y éstos satisfacen la condición:

$$C(N_1) * C(N_2) = C(N_1 + N_2),$$

entonces C(N) debe ser el residuo módulo b de N en una forma codificada, siendo b el número de símbolos redundantes distintos, y * la suma módulo b.

De este teorema concluímos que los códigos separados deben ser de la forma $(|N|_b, N)$. Peterson obtuvo este resultado para el conjunto infinito de los números enteros, pero en la práctica, N_1 y N_2 van a ser números formados por una cantidad finita de dígitos. N_1 y N_2 pueden tratarse como elementos de Z_m, y su suma como la suma módulo m: $|N_1 + N_2|_m$. Según trabajemos en aritmética de complemento a 1 o de complemento a 2, m será de la forma 2^n o 2^n-1 respectivamente.

Rao (Rao 1974) generalizó los resultados obtenidos por Peterson:

Lema 3.5.

Para cualquier N es $||N|_x|_y = |N|_y$ si y solo si x es múltiplo de y.

Teorema 3.15.

Sean N_1 y N_2 elementos de Z_m, y sea + la suma en Z_m. Supongamos que C(N) no necesita para su representación mas dígitos que N. Entonces se verifica:

$$C(N_1 + N_2) = C(N_1) * C(N_2)$$

si y solo si para algun b se cumplen las tres condiciones siguientes:

1) $C(N) = |N|_b$

2) * es la suma módulo b.

3) m es múltiplo de b.

Veamos a continuación algunos ejemplos.

Tomemos el anillo Z_m de los números enteros con $m = 2^{10} - 1$ (esto significa que representamos los enteros con 10 bits y trabajamos en aritmética de complemento a 1). Para formar un código de residuos $(|N|_b, N)$ que sea cerrado respecto de la suma, debemos elegir b de modo que m sea múltiplo de b. Así pues, podemos elegir entre los siguientes: 3, 11, 31, 33, 93, 341, 1023. Si elegimos b = 1023 el residuo se convierte en una mera duplicación del dato a codificar.

Si tomamos el anillo Z_m con $m = 2^{10}$ (trabajamos en aritmética de complemento a 2), cualquier código de residuos $(|N|_b, N)$ deberá tener b de tal modo que 2^{10} sea múltiplo de b. Por lo tanto, b será también una potencia de 2 , con lo cual $|N|_b$ es simplemente la duplicación de los bits menos significativos de N. Es evidente que este código no podrá detectar errores en los bits mas significativos de N salvo en el caso en que b = m, que no es eficiente.

3.5.2.3.1. Detección de errores con códigos separados.

Hemos visto que si trabajamos con un código de residuos, la parte de datos y el residuo se procesan en unidades independientes. Esto quiere decir que para efectuar la suma de dos palabras código utilizamos un sumador para los datos y otro para los residuos. Si el código es cerrado respecto de la suma (m es múltiplo de b), las salidas de ambos sumadores S y R deben formar una palabra código. Por tanto, si queremos detectar si se ha producido o no error, bastará con comprobar si la salida es una palabra código. Así, la función del detector de error (figura 3.6) será verificar la relación existente entre ambas salidas, y podrá consistir en un generador de residuo y un circuito comparador.

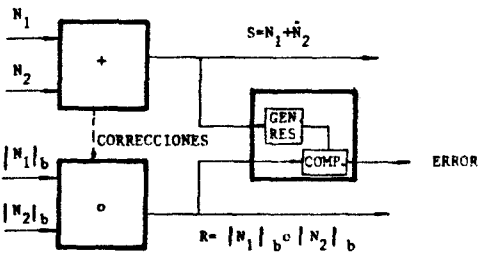


Figura 3.6

Si m no es múltiplo de b, el código no es cerrado respecto de la suma para todos los valores N_1 y N_2, y por tanto, S y R no siempre forman una palabra código. Sin embargo, podemos aplicar algunas correcciones para lograr que, siempre que no se haya producido error, S y R formen una palabra código.

Al construir un código de residuos es importante elegir la base b de modo que puedan detectarse los errores mas probables (en principio los errores simples). Distinguimos dos casos:

-Error en el sumador de datos. La salida del sumador es

$S_o = S_e + E = N_1 + N_2 + E$, donde E representa el error. El error E es un elemento de U(m,d). El detector de error calcula el residuo módulo b de S_o, que es:

$$|S_o|_b = |S_e + E|_b$$

que se compara con $R = |S_e|_b$ y se genera una señal de error si es $|E|_b \neq 0$.

-Error en el sumador de residuos. Sea $R_o = |R_e + e|_b$ la salida errónea del sumador de residuos, donde e es el error. R_o se compara con

$|S|_b$, y se detecta error cuando $|e|_b \neq 0$. En otras palabras, e puede detectarse si no es múltiplo de b.

Un problema de esta estructura de detección de errores es que si se produce un error, es imposible saber si ha fallado el procesador de datos o el de residuos.

3.5.2.3.2. Corrección de errores con códigos separados.

Hemos visto que con los códigos de residuos podemos detectar errores, pero no corregirlos. A fin de dotar a éstos de capacidad de corrección de errores, podemos hacer una generalización de estos códigos introducida por Rao (Rao 1974) que nos lleva a los códigos multirresiduo. Un dato X de Z_{m0}, podemos codificarlo en forma multirresiduo como un vector de t+1 componentes:

$$X \dashrightarrow (X, |X|_{m1}, |X|_{m2}, \ldots, |X|_{mt}) = (X, X_1, \ldots, X_t)$$

donde $X_i = |X|_{mi}$ es el residuo de X módulo m_i, y m_i para $1 \leq i \leq t$ son las bases de residuo. Normalmente se eligen como bases números primos entre sí.

Con este tipo de códigos, las operaciones aritméticas entre palabras código se ejecutan componente a componente. Por ejemplo, la suma de dos palabras código X e Y es:

$$X + Y = (|X + Y|_{m0}, |X_1 + Y_1|_{m1}, \ldots, |X_t + Y_t|_{mt})$$

Las unidades necesarias para ejecutar todas estas sumas son independientes entre sí, de modo que no hay transferencias de acarreo entre unas y otras. De este modo, los posibles errores que se produzcan en una de las unidades no pueden contaminar al resto. Uno de los códigos que encuentran mayor aplicación en unidades aritméticas es el código birresiduo.

3.5.2.3.3. Códigos birresiduo.

Un código birresiduo es un código separado con dos residuos. Un dato N de Z_{m0} se codifica mediante la terna (N,N_1,N_2), siendo $N_1 = |N|_{m1}$ y $N_2 = |N|_{m2}$. Formalizando la definición:

Una terna (x,y,z) se llama palabra código birresiduo respecto de las bases m_1 y m_2 si y solo si:

$$y = |x|_{m1}, \quad z = |x|_{m2}.$$

El síndrome de la terna (x,y,z) respecto de las bases m_1 y m_2, denotado por S(x,y,z) es un par (S_1,S_2), donde:

$$S_1 = |x - y|_{m1}, \text{ y } S_2 = |x - z|_{m2}$$

Deducimos de estas dos definiciones que una terna (x,y,z) de enteros es una palabra código birresiduo respecto de las bases m_1 y m_2 si su síndrome S(x,y,z) respecto de m_1 y m_2 es igual a (0,0).

Supongamos que se produce un error al procesar uno de los componentes de la palabra código. Podemos distinguir tres casos: Error en el dato, error en el primer residuo y error en el segundo residuo.

-Error en el dato. La palabra errónea será (x",y,z) con $x" = |x + E|_{m0}$. Si calculamos el síndrome $S(x,y,z) = (S_1,S_2)$, será:

$$S_1 = ||x + E|_{m0} - y|_{m1}; \; S_2 = ||x + E|_{m0} - z|_{m2}$$

Si las dos bases de residuo m_1 y m_2 son divisores del rango de representación m_0 (el código es cerrado), entonces, el síndrome es:

$$S(x",y,z) = (S_1,S_2) = (|E|_{m1},|E|_{m2}) = S(E,0,0)$$

Unicamente queda indetectado el error (y por lo tanto no puede co-

rregirse) si para $E \neq 0$ se verifica:

$$|E|_{m1} = |E|_{m2} = 0$$

Esto ocurre unicamente en el caso de que E sea múltiplo de m_1 y de m_2.

-Error en el primer residuo. La palabra errônea es ahora (x,y",z), donde $y" = |y + e|_{m1}$. El síndrome es:

$$S(x,y",z) = (|x - y"|_{m1}, |x - z|_{m2}) = (|-e|_{m1}, 0)$$

Solamente queda indetectado el error si para $e \neq 0$ es $|-e|_{m1} = 0$, caso que no puede darse porque no puede ser $e \geqslant m_1$.

-Error en el segundo residuo. Ahora la palabra errônea es (x,y,z"), donde es $z" = |z+e|_{m2}$. El síndrome es:

$$S(x,y,z") = (|x-y|_{m1}, |x-z"|_{m2}) = (0, |-e|_{m2})$$

Cuando falla uno de los dos residuos, el síndrome tiene exactamente un componente no nulo. De este modo, si se eligen adecuadamente las bases m_1 y m_2, de forma que para cualquier error en el dato $E \in U(m_0,d)$ sea $|E|_{m1} \neq 0$ y $|E|_{m2} \neq 0$, el síndrome de la palabra errônea tendrá sus dos componentes no nulos. Esto nos permite localizar perfectamente el error.

Si suponemos que en el mismo instante no puede fallar mas que una de las tres unidades, el síndrome localizará unívocamente la unidad que ha fallado:

a) $S_1 \neq 0$ $S_2 \neq 0$ Error en el dato.

b) $S_1 \neq 0$ $S_2 = 0$ Error en el primer residuo.

c) $S_1 = 0$ $S_2 \neq 0$ Error en el segundo residuo.

d) $S_1 = 0$ $S_2 = 0$ No hay error.

Si se localiza el error en uno de los dos residuos, podemos efectuar la corrección calculando de nuevo el residuo tomando como base de partida la parte de dato del resultado obtenido. Si lo que ha fallado es la parte de dato, seremos capaces de determinar el error partiendo del síndrome (S_1,S_2), siempre que hayamos elegido las bases m_1 y m_2 de modo que exista una correspondencia unívoca entre el conjunto de los síndromes que tienen sus dos componentes no nulas y el conjunto de los errores que queremos corregir (estos serán aquellos cuya probabilidad de ocurrencia sea mayor).

La forma de trabajo mas cómoda con estos códigos se produce cuando se cumplen las tres condiciones siguientes:

1) El rango de representación es $m_0 = 2^n - 1$, lo que significa que la unidad aritmética (para datos) tiene un ancho de n bits y además trabaja en aritmética de complemento a 1.

2) Las bases de residuos se toman de la forma $m_i = 2^{c_i} - 1$. De este modo, como veremos a continuación, tanto la codificación como la generación del síndrome son sencillas de realizar.

3) Los operadores de los residuos tienen longitud c_1 y c_2 respectivamente, que deben ser divisores de n.

Con estas condiciones se logra automáticamente que el código sea cerrado respecto de la suma. Si no se cumplen, tendremos que introducir algunas correcciones en el proceso de los residuos, como veremos en el siguiente capítulo.

Ahora el problema que se plantea es la elección de las bases de residuos m_1 y m_2 adecuadas. Rao (Rao 1974) demostró el siguiente teorema.

Teorema 3.16.

Sean $m_0 = 2^n - 1$, $m_1 = 2^a - 1$ y $m_2 = 2^b - 1$. Los síndromes correspondientes a los errores $E \in V(m_0,1)$ en el dato son todos distintos si $n = MCM(a,b)$, y no lo son si $n > MCM(a,b)$.

Este teorema nos permite elegir las bases m_1 y m_2 de modo que el código sea cerrado y tenga capacidad para corregir cualquier tipo de error simple en el dato.

3.5.2.3.4. Correspondencia entre códigos AN y códigos separados.

Podemos establecer una relación entre los códigos AN y los códigos separados. Consideremos un código AN cuyo rango de representación sea $M = Am$, y su generador A es de la forma

$$A = \prod_{i=1}^{t} m_i,$$

con los m_i primos entre sí. Asimismo, consideremos un código multirresiduo con $m_0 = Am$ y con bases de residuo m_i.

Hay una correspondencia biunívoca entre los errores en un código AN y los errores en la parte de dato de un código multirresiduo. Para un código separado, el síndrome de $X = (x, x_1, x_2, \ldots, x_t)$ es

$S(X) = (S_1, S_2, \ldots, S_t)$, con

$S_i = |x - x_i|_{mi}$ para $i = 1,2,\ldots t$.

Sea $Z = (z, z_1, \ldots, z_t)$ el resultado correcto de la suma de dos palabras código multirresiduo. Si se ha producido un error E en el dato, entonces:

$$S(z + E) = S(|z + E|_{m0}, z_1, \ldots, z_t) = (|E|_{m1}, |E|_{m2}, \ldots, |E|_{mt})$$

Lema 3.6.

Sea A de la forma:

$$A = \prod_{i=1}^{t} m_i,$$

con los m_i primos entre sí. Dado un código AN capaz de corregir todos los errores $E \in U(M,d)$ para $M = Am$, existe un código separado con t residuos que tiene bases $m_1, m_2, \ldots, m_3$, rango de representación $m_0 = M$, y existe un síndrome distinto para cada error $E \in U(m_0,d)$ en los datos. Recíprocamente, si tenemos un código multirresiduo con síndromes distintos para cada error $E \in U(m_0,d)$ en los datos, existe un código AN que tiene las mismas propiedades de corrección de error.

Demostración.

La correspondencia entre los códigos AN y los multirresiduo es como sigue. Cualquier error E en el código AN tiene como síndrome $|E|_A$. Cualquier error en la parte de datos de un código multirresiduo tiene como síndrome $S(E) = (|E|_{m1},\ldots,|E|_{mt})$, donde A es múltiplo de cada uno de los m_i. Por tanto, m_0 es múltiplo de cada m_i. Si los síndromes de todos los errores $E \in U(M,d)$ son distintos, entonces son elementos distintos del anillo de los enteros módulo A. Así pues, los síndromes de todos los errores $E \in U(m_0,d)$ son distintos.

3.6. Códigos de bajo costo.

Tanto con los códigos AN como con los gAN y los de residuos, debemos encontrar un modo facil para efectuar tanto la decodificación como el cálculo del síndrome.

Para códigos AN y gAN, la codificación consiste en un producto por A, y el cálculo del síndrome consiste en calcular el residuo módulo A del resultado. Para códigos de residuos, tanto la codificación como el cálculo del síndrome consiste en calcular un residuo módulo una cierta base m_i.

En general, este cálculo es complejo y lento, pues necesita la ejecución de una división. Ahora bien, si elegimos la base de la forma $A = 2^c - 1$ (Avizienis 1961), el residuo módulo A de un número N se puede obtener sin necesidad de ejecutar divisiones. Tomemos como ejemplo n = 11010110011. El cálculo del residuo de N módulo 3 ($|N|_3$) podemos efectuarlo mediante sumas repetidas de rodajas de dos bits realimentando el acarreo (figura 3.7).

N = 1|10|10|11|00|11

```
   11
  00
   11
  11
1)10
   1
   11
   10
1)01
   1
   10
   10
1)00
   1
   01
   01
   10
```

$|N|_3 = |11010110011|_3 = |1715|_3 = 2 = 10$

Figura 3.7.

Podemos obtener un método general para el cálculo del residuo módulo $r^c - 1$ de un entero en base r con n dígitos.

Sea $X = X_{n-1}, \ldots, X_0$ *un entero con n dígitos en base r.* Dividimos dicho entero en l rodajas de longitud c, de modo que se verifica:

$$(l - 1)\, c < n \leqslant lc$$

y llamamos a estas rodajas $B_{l-1}, \ldots, B_1, B_0$. Entonces tenemos:

$$X = \sum_{j=0}^{l-1} B_j\, r^{jc}, \text{ donde } 0 \leqslant B_j < r^c$$

Así pues:

$$|X|_A = |X|_{r^c-1} = \left|\sum_{j=0}^{l-1} B_j\, r^{jc}\right|_{r^c-1}$$

Como $|r^{jc}|_{r^c-1} = 1$ para todo $j \geqslant 0$, resulta que:

$$|X|_A = \left|\sum_{j=0}^{l-1} B_j\right|_{r^c-1}$$

Así pues queda demostrado que podemos calcular el residuo módulo $r^c - 1$ sumando las rodajas en sumadores módulo $r^c - 1$.

A estos códigos en los que se toma la base de la forma $A = r^c - 1$ se les llama "códigos de bajo costo".

CAPITULO IV.

CODIGOS MULTIRRESIDUO Y OPERACIONES ELEMENTALES.

Establecimos en el capítulo anterior que los códigos multirresiduo son cerrados respecto de la suma y la diferencia si cada una de las bases de residuo m_i es divisor del rango de representación m_0.

Si tomamos cada m_i de la forma $m_i = 2^{c_i} - 1$ (sencillez de codificación), para que el código sea cerrado respecto de la suma, debe ser $m_0 = 2^n - 1$, con n múltiplo de c_i para cada i. Esto significa que debemos trabajar en complemento a uno y que se nos imponen unas restricciones muy fuertes en la elección de las bases de residuo.

Supongamos por ejemplo, que queremos construir una unidad de proceso para 16 bits de datos protegida mediante un código birresiduo. Podríamos en principio utilizar un código con $m_1 = 15$ y $m_2 = 31$ que (como veremos) permite utilizar datos de hasta 20 bits. Pero si $m_0 = 2^{16} - 1$, $m_1 = 2^4 - 1$ y $m_2 = 2^5 - 1$, el código no es cerrado respecto de la suma porque 16 no es múltiplo de 5.

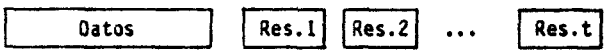
Del mismo modo, estos códigos no son cerrados respecto de operaciones tales como desplazamientos, lógicas, etc. Este problema se ha tratado en la literatura (Rao 1974), pero solamente llegando a soluciones que no tienen una facil implantación física.

En este capítulo planteamos y demostramos 8 teoremas inéditos cuya aplicación permite obligar a que un código multirresiduo sea cerrado respecto de todas las operaciones elementales, trabajando tanto en aritmética de complemento a uno como en aritmética de complemento a dos, y sin necesidad de que las bases de residuo sean divisores del rango de representación. Además, la solución propuesta en esta Tesis tiene, como veremos, una facil

implantación física.

Partiremos de las siguientes premisas:

Si trabajamos con un código multirresiduo, las palabras código están formadas por la concatenación de un bloque de datos y varios bloques correspondientes a los residuos módulo m_i de dichos datos, es decir,



Una operación entre palabras código equivale a una operación f_d entre los bloques de datos y t operaciones f_r entre los bloques de residuo correspondientes. El resultado de la operación será una palabra código (y por tanto el código será cerrado respecto de dicha operación) si el resultado de la operación f_r sobre los residuos módulo m_i es igual al residuo módulo m_i del resultado de la operación f_d sobre los bloques de datos. De este modo, nuestro problema es encontrar la operación f_r a ejecutar sobre los bloques de residuo, que corresponda a cada una de las operaciones elementales f_d sobre los datos, de modo que la palabra resultante sea una palabra código.

A continuación hacemos un estudio detallado para cada una de las operaciones elementales f_d. Trabajaremos con un solo residuo, puesto que los resultados que se obtienen son inmediatamente extensibles al caso multirresiduo. Tomaremos como base de dicho residuo $m_i = A$.

4.1 *Correspondencia entre operaciones.*

Supongamos que a la entrada de la unidad de proceso se presentan en el instante t dos palabras código (X_1,R_1) y (X_2,R_2), y las señales de control indican que hay que ejecutar la operación f_d. La salida del procesador viene dada en el instante t + 1 por S y Q de modo que:

$$S(t+1) = f_d(X_1, X_2), \text{ y}$$

$$Q(t+1) = f_r(R_1, R_2)$$

Tanto S como Q serán función de ninguno (Preset, Clear), uno (operación unaria) o ambos operandos (operación binaria), dependiendo del código f_d.

Consideramos que los datos tienen una longitud de n bits, y que el procesador puede ejecutar operaciones aritméticas y lógicas sobre éstos. Así pues, un dato X será $X = (X_{n-1}, X_{n-2}, \ldots X_0)$, y su valor vendrá determinado por la expresión:

$$V(X) = \sum_{i=0}^{n-1} X_i(t)\ 2^i$$

El rango de representación será $m = 2^n$ si trabajamos en aritmética de complemento a dos y $2^n - 1$ si trabajamos en aritmética de complemento a uno.

El residuo tendrá como base A de la forma $A = 2^a - 1$, lo que facilita la decodificación y el cálculo del síndrome.

Las operaciones elementales que vamos a considerar son las siguientes:

Suma.
Negación.
Diferencia.
Desplazamientos.
"Y" lógico.
"O" lógico.
"O" exclusivo lógico.

4.1.1. Operaciones aritméticas.

4.1.1.1. Suma.

Supongamos que queremos sumar las dos palabras código (X_1,R_1) y (X_2,R_2).

Teorema 4.1.

Sea la suma la operación f_d a ejecutar sobre los datos X_1 y X_2, y sea C_n el acarreo mas allá del bit n producido en esta suma. El resultado será una palabra código si sobre los residuos R_1 y R_2 se ejecuta la operación f_r siguiente:

$$f_r(R_1,R_2) = ||R_1 + R_2|_A - C_n|m|_A|_A$$

Demostración.

Debido a que el procesador de datos tiene una longitud finita n, la suma siempre se ejecutará módulo m, es decir, que se desprecia el acarreo C_n que se produce (aunque se almacene en el flag correspondiente). Este acarreo tiene valor m, y despreciarlo equivale a restar del valor entero de la suma la cantidad m. De este modo, el valor de la suma de los datos X_1 y X_2 es:

$$V(S) = V(X_1 + X_2) = V(X_1) + V(X_2) - mC_n$$

Si calculamos el residuo módulo A de este valor, queda:

$$R = |V(X_1 + X_2)|_A = |V(X_1)|_A + |V(X_2)|_A - |mC_n|_A$$

Como $|V(X_i)|_A = R_i$, nos queda:

$$f_r(R_1,R_2) = ||R_1 + R_2 - C_n|m|_A|_A \quad \text{c.q.d.}$$

Consecuencia inmediata de este teorema es que la operación f_r a ejecutar sobre los residuos cuando sobre los datos se ejecuta la suma módulo m, es la suma módulo A, seguida de una corrección que consiste en restar módulo A la cantidad $C_n|m|_A$.

Veamos el caso particular en que n es múltiplo de a.

Si trabajamos en aritmética de complemento a uno, es $m = 2^n - 1$, y por tanto, m es múltiplo de A, por lo que $|m|_A = 0$. Así pues, en este caso no es necesaria la corrección, y la operación f_r correspondiente a la suma módulo m sobre los datos es la suma módulo A de los residuos.

Trabajando en aritmética de complemento a dos, es $m = 2^n$, y $|m|_A = |2^n|_{2^a-1} = 1$. Por lo tanto, en este caso, después de efectuar la suma módulo A de los residuos, se debe restar C_n al resultado para que el código permanezca cerrado.

4.1.1.2. Negación.

Supongamos que queremos negar la palabra código (X,R). Nos apoyamos en el siguiente teorema:

Teorema 4.2.

Sea la negación la operación f_d a ejecutar sobre el dato X. El resultado será una palabra código si sobre el residuo R se ejecuta la operación f_r siguiente:

$$f_r(R) = |\bar{R} + |m|_A|_A$$

Demostración.

La negación de un dato se ejecuta restando éste del rango de representación. Sea $\bar{X}$ el negado de X. Así,

$$V(\bar{X}) = m - V(X)$$

Si calculamos el residuo módulo A, queda:

$$|V(\bar{X})|_A = |m - V(X)|_A = ||m|_A - |V(X)|_A|_A =$$

$$= ||m|_A - R|_A = ||m|_A + \bar{R}|_A. \text{ De modo que:}$$

$$f_r(R) = |\bar{R} + |m|_A|_A. \qquad \text{c.q.d.}$$

Con este teorema vemos que la operación f_r a ejecutar sobre el residuo cuando sobre el dato se ejecuta una negación, es una negación seguida de una corrección consistente een sumar módulo A la cantidad $|m|_A$.

4.1.1.3. Diferencia.

Supongamos que queremos calcular la diferencia entre las dos palabras código (X_1,R_1) y (X_2,R_2).

Teorema 4.3.

Sea la diferencia la operación f_d a ejecutar sobre los datos X_1 y X_2, y sea C_n el acarreo mas alla del bit n producido en ésta. El resultado será una palabra código si sobre los residuos R_1 y R_2 se ejecuta la operación f_r siguiente:

$$f_r(R_1,R_2) = |R_1 - R_2 + (1 - C_n)|m|_A|_A$$

Demostración.

La diferencia entre dos datos X_1 y X_2 se ejecuta sumando a X_1 el negado de X_2, es decir,

$$V(X_1 - X_2) = V(X_1 + \bar{X}_2) = V(X_1) + V(\bar{X}_2) - mC_n$$

Calculando el residuo módulo A queda:

$$|V(X_1 - X_2)|_A = |V(X_1) + V(\overline{X}_2) - mC_n|_A =$$

$$= ||V(X_1)|_A + |V(\overline{X}_2)|_A - C_n|m|_A|_A =$$

$$= |R_1 + |m|_A + \overline{R}_2 - C_n|m|_A|_A. \text{ Por tanto,}$$

$$f_r(R_1,R_2) = |R_1 - R_2 + (1 - C_n)|m|_A|_A. \quad \text{c.q.d.}$$

Vemos con este teorema que la operación f_r que debe ejecutarse sobre los residuos cuando sobre los datos se ejecuta una diferencia, es la diferencia módulo A, seguida de una corrección consistente en sumar módulo A al resultado la cantidad $(1 - C_n)|m|_A$.

4.1.2. Desplazamientos.

Obtenemos en este apartado las operaciones f_r correspondientes al desplazamiento de un dato $X = (X_{n-1}, X_{n-2}, \ldots, X_1, X_0)$ tanto a la derecha como a la izquierda, introduciendo por su derecha en el primer caso un bit X_{-1}, y por su izquierda en el segundo caso un bit X_n. De este modo cubrimos todos los posibles tipos de desplazamiento, ya sean circulares (rotaciones), aritméticos o lógicos. A continuación damos una lista de todos estos desplazamientos, y sus correspondientes operaciones f_r sobre los residuos.

Lema 4.1.

Sea un dato $X = (X_{n-1}, X_{n-2}, \ldots X_0)$ de longitud n. Si el rango de representación es $m = 2^n - 1$, el valor del resultado de rotar a la derecha dicho dato viene dado por:

$$|\tfrac{1}{2} V(X)|_m$$

Demostración.

Partimos del dato X en el instante t:

$$X(t) = X_{n-1}(t),\ X_{n-2}(t),\ \ldots\ ,\ X_0(t)$$

El resultado de la rotación será en el instante t + 1:

$$X(t+1) = X_{n-1}(t+1),\ X_{n-2}(t+1),\ \ldots\ ,\ X_0(t+1)$$

Para ejecutar la rotación hacemos:

$$X_{n-1}(t+1) = X_0(t) \quad y$$

$$X_j(t+1) = X_{j+1}(t),\ \text{para } j = 0,1,\ldots,n-2$$

De este modo, el valor del resultado X(t+1) es:

$$V(X(t+1)) = \sum_{i=0}^{n-1} X_i(t+1)\ 2^i = X_0(t)2^{n-1} + \sum_{i=0}^{n-2} X_{i+1}(t)\ 2^i =$$

$$= X_0(t)\ 2^{n-1} + \sum_{i=0}^{n-2} X_{i+1}(t)\ \ \frac{1}{2}\ \ 2^{i+1}$$

Haciendo j = i + 1, tenemos:

$$V(X(t+1)) = X_0(t)\ 2^{n-1} + \frac{1}{2} \sum_{j=1}^{n-1} X_j(t)\ 2^j + \frac{1}{2} X_0(t) - \frac{1}{2} X_0(t) =$$

$$= X_0(t)\ 2^{n-1} + \frac{1}{2} \sum_{j=0}^{n-1} X_j(t)\ 2^j - \frac{1}{2} X_0(t)$$

Como $V(X(t)) = \sum_{j=0}^{n-1} X_j(t)\ 2^j$, queda:

$$V(X(t+1)) = X_0(t)\ 2^{n-1} + \frac{1}{2} V(X(t)) - \frac{1}{2} X_0(t) =$$

$$= \frac{1}{2} V(X(t)) + (2^n - 1)\ \frac{1}{2} X_0(t)$$

Como trabajamos en módulo m, será:

$$|V(X(t+1))|_m = |\tfrac{1}{2} V(X(t)) + (2^n - 1)\ \tfrac{1}{2} X_0(t)|_m$$

Por ser $m = 2^n - 1$, es $|2^n - 1|_m = 0$, con lo cual:

$$|V(X(t+1))|_m = |\tfrac{1}{2} V(X(t))|_m. \quad \text{c.q.d.}$$

Así pues, acabamos de demostrar que el valor de una rotación a la derecha sobre un dato de n bits cuando $m = 2^n - 1$ es $|\frac{1}{2} V(X)|_m$. Este resultado nos servirá de apoyo para demostrar el teorema 4.4, que nos dice cual es la operación f_r correspondiente a un desplazamiento genérico a la derecha.

Teorema 4.4.

Tomemos un dato $X = X_{n-1}, X_{n-2}, \ldots, X_0$, cuyo residuo módulo A es R. Sea un desplazamiento a la derecha de este dato, introduciendo por su izquierda el bit X_n, la operación f_d a ejecutar sobre él. Para que el resultado sea una palabra código, la operación f_r que debemos ejecutar sobre el residuo es una rotación a la derecha, seguida de la suma módulo A de la cantidad:

$$|X_n|2^{n-1}|_A - X_0\ 2^{a-1}|_A$$

Demostración.

Partimos del dato X en el instante t:

$$X(t) = X_{n-1}(t), \ldots, X_0(t).$$

Ejecutamos un desplazamiento a la derecha de X introduciendo por su izquierda el bit $X_n(t)$. El resultado del desplazamiento es:

$$X(t+1) = X_{n-1}(t+1), X_{n-2}(t+1), \ldots, X_0(t+1), \text{ donde}$$

$$X_{n-1}(t+1) = X_n(t), \text{ y}$$

$X_j(t+1) = X_{j+1}(t)$ para $j = 0, 1, \ldots, n-2$

El valor del resultado será:

$$V(X(t+1)) = \sum_{i=0}^{n-1} X_i(t+1)\, 2^i =$$

$$= X_n(t)\, 2^{n-1} + \sum_{i=0}^{n-2} X_{i+1}(t)\, 2^i =$$

$$= X_n(t)\, 2^{n-1} + \frac{1}{2} \sum_{j=1}^{n-1} X_j(t)\, 2^j + \frac{1}{2} X_0(t) - \frac{1}{2} X_0(t) =$$

$$= X_n(t)\, 2^{n-1} + \frac{1}{2} \sum_{j=0}^{n-1} X_j(t)\, 2^j - \frac{1}{2} X_0(t)$$

Como es $V(X(t)) = \sum_{j=0}^{n-1} X_j(t)\, 2^j$, tenemos:

$$V(X(t+1)) = \frac{1}{2} V(X(t)) + X_n(t)\, 2^{n-1} - \frac{1}{2} X_0(t).$$

Calculando el residuo módulo A obtenemos:

$$f_r(R) = \left|\frac{1}{2} V(X(t)) + X_n(t)\, 2^{n-1} - \frac{1}{2} X_0(t)\right|_A =$$

$$= \left|\left|\frac{1}{2} V(X(t))\right|_A + X_n(t)\, \left|2^{n-1}\right|_A - X_0(t)\, \left|\frac{1}{2}\right|_A\right|_A$$

Calculemos ahora el valor de cada uno de los términos.

a) $\left|\frac{1}{2} V(X(t))\right|_A = \left|\frac{1}{2}\right|_A \left|V(X(t))\right|_A =$

$$= \left|\frac{1}{2}\right|_A R(t) = \left|\frac{1}{2} R(t)\right|_A$$

Tomando $A = 2^a - 1$, como R tiene a bits, aplicando el lema 4.1, resulta que $\left|\frac{1}{2} R(t)\right|_A$ es el valor del resultado de ejecutar una rotación a la derecha sobre el residuo.

b) $X_0(t)\, \left|\frac{1}{2}\right|_A$;

$\left|\frac{1}{2}\right|_A = u$ es equivalente a la congruencia $2u = 1 \pmod{A}$.

Tomando $A = 2^a-1$, obtenemos que $2u = 2^a$. Por tanto, es $u = 2^{a-1}$. Así pues, tenemos:

$$X_0(t)\ |\tfrac{1}{2}|_A = X_0(t)\ 2^{a-1}.$$

c) $X_n(t)\ |2^{n-1}|_A$.

No tiene simplificación salvo en el caso particular en que n sea múltiplo de a. Entonces es $|2^{n-1}|_A = 2^{a-1}$. c.q.d.

Por lo tanto, acabamos de demostrar que la operación f_r que debemos ejecutar sobre el residuo cuando sobre el dato ejecutamos un desplazamiento a la derecha (introduciendo por la izquierda un bit X_n), es una rotación a la derecha sobre el residuo, seguida de la suma módulo A de la cantidad:

$$|X_n\ |2^{n-1}|_A - X_0\ 2^{a-1}|_A$$

Lema 4.2.

Sea un dato $X = X_{n-1}, X_{n-2}, \ldots, X_0$ de longitud n. Si el rango de representación es $m = 2^n-1$, el valor del resultado de rotar a la izquierda dicho dato viene dado por:

$$|2\ V(X)|_m$$

Demostración.

Sea el dato X en el instante t $X(t) = X_{n-1}(t), \ldots, X_0(t)$. El resultado de una rotación a la izquierda será:

$X(t+1) = X_n(t+1), \ldots, X_0(t+1)$, donde:

$X_0(t+1) = X_{n-1}(t)$, y

$X_j(t+1) = X_{j-1}(t)$ para $j = 1, 2, \ldots, n-1$.

De este modo, el valor de X(t+1) es:

$$V(X(t+1)) = \sum_{i=0}^{n-1} X_i(t+1)\, 2^i =$$

$$= \sum_{i=1}^{n-1} X_{i-1}(t)\, 2^i + X_{n-1}(t).$$

Haciendo j = i-1, tenemos:

$$V(X(t+1)) = \sum_{j=0}^{n-2} X_j(t)\, 2^{j+1} + X_{n-1}(t) + X_{n-1}(t)\, 2^n - X_{n-1}(t)\, 2^n =$$

$$= \sum_{j=0}^{n-2} X_j(t)\, 2^{j+1} + X_{n-1}(t)\, 2^n - X_{n-1}(t)\, (2^n-1) =$$

$$= \sum_{j=0}^{n-1} 2X_j(t)\, 2^j - X_{n-1}(t)(2^n-1)$$

Como es $V(X(t)) = \sum_{j=0}^{n-1} X_j(t)\, 2^j$, tenemos:

$$V(X(t+1)) = 2\,V(X(t)) - X_{n-1}(t)(2^n-1).$$

Como estamos trabajando módulo m, será:

$$|V(X(t+1))|_m = |2\,V(X(t)) - X_{n-1}(t)(2^n-1)|_m =$$

$$= ||2\,V(X(t))|_m - |X_{n-1}(t)(2^n-1)|_m|_m.$$

Como es $m = 2^n-1$, es $|2^n-1|_m = 0$. Por tanto, queda:

$$|V(X(t+1))|_m = |2\,V(X(t))|_m. \quad \text{c.q.d.}$$

Este lema nos sirve de base para demostrar el teorema 4.5, que nos dice cual es la operación f_r correspondiente a un desplazamiento genérico a la izquierda.

Teorema 4.5.

Tomemos un dato $X = X_{n-1}, X_{n-2}, \ldots, X_0$. Sea un desplazamiento a la izquierda de dicho dato, introduciendo por su derecha el bit X_{-1}, la operación f_d a ejecutar sobre él. Para que el resultado sea una palabra código, la operación f_r que debemos ejecutar sobre el residuo es una rotación a la izquierda, seguida de la suma módulo A de la cantidad:

$$|X_{-1} - X_{n-1}(t)|2^n|_A|_A.$$

Demostración.

Sea X en el instante t $X(t) = X_{n-1}(t), \ldots, X_0(t)$. Ejecutamos un desplazamiento a la izquierda de X introduciendo por su derecha el bit $X_{-1}(t)$. El resultado de este desplazamiento será:

$X(t+1) = X_{n-1}(t+1), X_{n-2}(t+1), \ldots, X_0(t+1)$, donde:

$X_0(t+1) = X_{-1}(t)$, y

$X_j(t+1) = X_{j-1}(t)$ para $j = 1, 2, \ldots, n-1$

El valor del resultado es:

$$V(X(t+1)) = \sum_{i=0}^{n-1} X_i(t+1)\, 2^i =$$

$$= \sum_{i=1}^{n-1} X_{i-1}(t)\, 2^i + X_{-1}(t)$$

$$= \sum_{j=0}^{n-2} X_j(t)\, 2^{j+1} + X_{-1}(t) + X_{n-1}(t)\, 2^n - X_{n-1}(t)\, 2^n =$$

$$= \sum_{j=0}^{n-1} X_j(t)\, 2^{j+1} + X_{-1} - X_{n-1}(t)\, 2^n =$$

$$= 2 \sum_{j=0}^{n-1} X_j(t)\, 2^j + X_{-1}(t) - X_{n-1}(t)\, 2^n.$$

Como es $V(X(t)) = \sum_{j=0}^{n-1} X_j(t)\, 2^j$, entonces:

$$V(X(t+1)) = 2\ V(X(t)) + X_{-1}(t) - X_{n-1}(t)\ 2^n.$$

Si calculamos el residuo módulo A, tenemos:

$$f_r(R) = |V(X(t+1))|_A = |2\ V(X(t)) + X_{-1} - X_{n-1}(t)\ 2^n|_A =$$

$$= ||2\ V(X(t))|_A + X_{-1}(t) - X_{n-1}(t)\ |2^n|_A;$$

Pero como $|2\ V(X(t))|_A = |2|_A|V(X(t)|_A = |2R(t)|_A$, entonces:

$$f_r(R) = ||2\ V(R(t))|_A + X_{-1}(t) - X_{n-1}|2^n|_A|_A.$$

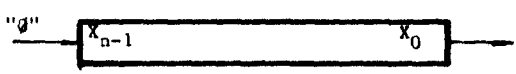
Como por el lema 4.2 $|2\ V(R(t))|_A$ es el valor del resultado de una rotación a la izquierda del residuo, la operación f_r que debemos ejecutar sobre el residuo es una rotación a la izquierda, seguida de la suma módulo A de la cantidad:

$$|X_{-1}(t) - X_{n-1}(t)|2^n|_A|_A.$$ c.q.d.

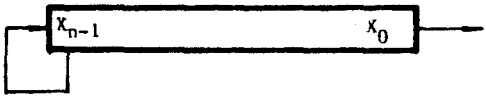
Los desplazamientos que podemos ejecutar sobre las palabras código son los siguientes:

Desplazamiento a la derecha:

- Lógico.

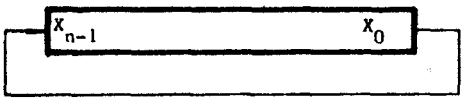


- Aritmético.

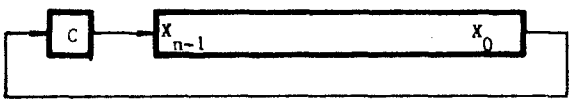


- Con acarreo.

- Circular.



- Circular con acarreo.



$2^{a-1} = 2^3 = 8$

$|2^n|_{15} = |2^{16}|_{15} = 1$

$|2^{n-1}|_{15} = |2^{15}|_{15} = 8$

Para la base $m_2 = 2^5 - 1 = 31$, es:

$2^{a-1} = 2^4 = 16$

$|2^n|_{31} = |2^{16}|_{31} = 2$

$|2^{n-1}|_{31} = |2^{15}|_{31} = 1$

Con estos valores construimos la tabla 4.1 en la que aparece cada tipo de desplazamiento y la cantidad a sumar al resultado de la rotación del residuo para que el código sea cerrado.

		Cantidad a sumar despues de rotación.	
		m = 15	m = 31
Desplazamiento a la derecha	Lógico	$8X_0$	$16X_0$
	Aritmético	$\|8X_{n-1}-8X_0\|_{15}$	$\|X_{n-1}-16X_0\|_{31}$
	Con acareo	$\|8C-8X_0\|_{15}$	$\|C-16X_0\|_{31}$
	Circular	-----	$\|X_0-16X_0\|_{31}$
	Circular c/c	$\|8C-8X_0\|_{15}$	$\|C-16X_0\|_{31}$
Desplazamiento a la izda.	lógico	$\|-X_{n-1}\|_{15}$	$\|-2X_{n-1}\|_{31}$
	Con acarreo	$\|C-X_{n-1}\|_{15}$	$\|C-2X_{n-1}\|_{31}$
	Circular	-----	$\|X_{n-1}-2X_{n-1}\|_{31}$
	Circular c/c	$\|C-X_{n-1}\|_{15}$	$\|C-2X_{n-1}\|_{31}$

Tabla 4.1.

Vemos entonces que el código puede hacerse cerrado respecto de

cualquier desplazamiento sin mas que sumar ciertas constantes al resultado de ejecutar una rotación sobre el residuo.

4.1.3. Operaciones lógicas.

Al tratar de utilizar un código de residuos para proteger las operaciones lógicas, nos encontramos con un problema importante: El código de residuos no es cerrado respecto de ninguna de ellas. Sin embargo, podemos utilizar los resultados del siguiente teorema.

Lema 4.3.

Sean X e Y dos vectores binarios de longitud n y sean $^{\circ}$, v y $\oplus$ respectivamente las operaciones "Y", "O" y "O" exclusivo. Entonces, se verifican las igualdades siguientes:

$$V(X \vee Y) = V(X) + V(Y) - V(X^{\circ}Y)$$

$$V(X \oplus Y) = V(X) + V(Y) - 2V(X^{\circ}Y)$$

Demostración.

Primero demostraremos que para un solo bit X_i e Y_i se verifica que:

$$X_i \vee Y_i = X_i + Y_i - X_i \circ Y_i \quad (1)$$

$$X_i \oplus Y_i = X_i + Y_i - 2X_i \circ Y_i \quad (2)$$

$X_i \vee Y_i$ queda definido por la siguiente tabla:

X_i	Y_i	$X_i \vee Y_i$
0	0	0
0	1	1
1	0	1
1	1	1

Extrapolamos a la función:

$$X_i \vee Y_i = a X_i + b Y_i + c X_i \cdot Y_i$$

Sustituyendo valores encontramos:

$1 = a$
$1 = b$
$1 = a + b + c \qquad c = -1$

De modo que:

$$X_i \vee Y_i = X_i + Y_i - X_i \cdot Y_i$$

La función "O" exclusivo se define por la tabla siguiente:

X_i	Y_i	$X_i \oplus Y_i$
0	0	0
0	1	1
1	0	1
1	1	0

Extrapolamos a la función:

$$X_i \oplus Y_i = a X_i + b Y_i - c X_i \cdot Y_i$$

Sustituyendo valores obtenemos:

$1 = a$

$1 = b$

$0 = a + b + c \qquad c = -2$

Así pues, se verifica que:

$$X_i \oplus Y_i = X_i + Y_i - 2\, X_i \cdot Y_i$$

Demostraremos ahora el lema.

En (1) multiplicamos ambos miembros por 2^i y sumamos desde $i = 0$ hasta $i = n-1$:

$$\sum_{i=0}^{n-1} (X_i \vee Y_i)\, 2^i = \sum_{i=0}^{n-1} X_i\, 2^i + \sum_{i=0}^{n-1} Y_i\, 2^i - \sum_{i=0}^{n-1} (X_i \cdot Y_i)\, 2^i,$$

O lo que es lo mismo:

$$V(X \vee Y) = V(X) + V(Y) - V(X \cdot Y)$$

Haciendo la misma operación con la ecuación (2) tenemos:

$$\sum_{i=0}^{n-1} (X_i \oplus Y_i)\, 2^i = \sum_{i=0}^{n-1} X_i\, 2^i + \sum_{i=0}^{n-1} Y_i\, 2^i - 2 \sum_{i=0}^{n-1} (X_i \cdot Y_i)\, 2^i,$$

Que resulta:

$$V(X \oplus Y) = V(X) + V(Y) - 2\, V(X \cdot Y). \quad \text{c.q.d.}$$

Mediante la aplicación de este lema, podemos obtener cualquiera de las operaciones lógicas a partir de una de ellas y la suma. Así pues, la estrategia propuesta para proteger las operaciones lógicas es aplicar este teorema para calcular el residuo resultante, usando para ello la duplica-

ción de una de las operaciones lógicas. Si además queremos proteger los operandos, duplicaremos dos de ellas. Esta duplicación tiene una realización práctica muy poco costosa.

Veamos ahora cual es la operación f_r que corresponde a cada una de las operaciones lógicas.

4.1.3.1. "Y" lógico.

Supongamos que queremos calcular el "Y" lógico entre dos operandos cuyas partes de dato son X_1 y X_2.

Teorema 4.6.

Sea el "Y" lógico la operación f_d a ejecutar sobre los datos X_1 y X_2. El resultado será una palabra código si sobre los residuos se ejecuta la siguiente operación f_r:

$$f_r(R_1,R_2) = |R_1 + R_2 - |V(X_1 \vee X_2|_A|_A$$

Demostración.

Por el lema 4.3 se verifica:

$V(X_1 \vee X_2) = V(X_1) + V(X_2) - V(X_1 * X_2)$. Así pues,

$V(X_1 * X_2) = V(X_1) + V(X_2) - V(X_1 \vee X_2)$.

Si calculamos el residuo módulo A:

$|V(X_1 * X_1)|_A = ||V(X_1)|_A + |V(X_2)|_A - |V(X_1 \vee X_2)|_A|_A$; Así se verifica que:

$$f_r(R_1,R_2) = |R_1 + R_2 - |V(X_1 \vee X_2)|_A|_A. \quad \text{c.q.d.}$$

Así pues, la operación correspondiente a un "Y" lógico entre dos datos, será la suma de los residuos de éstos seguida de una corrección consistente en restar módulo A la cantidad $|V(X_1 \vee X_2)|_A$, que calcularemos a partir de una duplicación del "O" lógico.

4.1.3.2. "O" lógico.

Queremos ahora calcular el "O" lógico de los datos X_1 y X_2.

Teorema 4.7.

Sea el "O" lógico la operación f_d a ejecutar sobre los datos X_1 y X_2. El resultado será una palabra código si la operación f_r que se ejecuta sobre los residuos es la siguiente:

$$f_r(R_1,R_2) = |R_1 + R_2 - |V(X_1 * X_2)|_A|_A$$

Demostración.

Por el lema 4.3 sabemos que:

$$V(X_1 \vee X_2) = V(X_1) + V(X_2) - V(X_1 * X_2)$$

Calculando el residuo módulo A queda:

$$f_r(R_1,R_2) = |R_1 + R_2 - |V(X_1 * X_2)|_A|_A \quad \text{c.q.d.}$$

Por tanto, la operación f_r correspondiente a un "O" lógico entre dos datos es la suma de los residuos de éstos seguida de una corrección que consiste en restar módulo A la cantidad $|V(X_1 * X_2)|_A$, que obtenemos de una duplicación del "Y" lógico.

4.1.3.3. "O" exclusivo.

Supongamos ahora que queremos calcular el "O" exclusivo de dos palabras código cuyas partes de dato son X_1 y X_2.

Teorema 4.8.

Sea el "O" exclusivo la operación f_d a ejecutar sobre los datos X_1 y X_2. El resultado será una palabra código si la operación f_r que se ejecuta sobre los residuos es la siguiente:

$$f_r(R_1,R_2) = |R_1 + R_2 - |2\ V(X_1 * X_2)|_A|_A$$

Demostración.

Por el lema 4.3 sabemos que:

$$V(X_1 \oplus X_2) = V(X_1) + V(X_2) - 2\ V(X_1 * X_2)$$

Calculando el residuo módulo A queda:

$$f_r(R_1,R_2) = |R_1 + R_2 - |2\ V(X_1 * X_2)|_A|_A.\ \text{c.q.d.}$$

Vemos pues que la operación f_r que corresponde a un "O" exclusivo de los datos es la suma de los residuos seguida de una corrección consistente en restar módulo A la cantidad $|2\ V(X_1 * X_2)|_A$.

4.2. Conclusión.

Con los nueve teoremas anteriores hemos establecido la operación f_r que hay que ejecutar sobre los residuos, correspondiente a cada una de las operaciones elementales que se ejecutan sobre los datos, de modo que el código permanezca cerrado respecto de éstas, es decir, que el residuo del resultado de una operación f_d sobre los datos sea igual al resultado de aplicar la correspondiente operación f_r sobre los residuos de dichos datos. De

este modo, si no se ha producido error, el síndrome será cero.

Cada operación f_r podemos dividirla en dos partes, una operación previa sobre los residuos, y una corrección, que consiste en sumar módulo A una cierta cantidad.

En la tabla 4.2 se resumen los resultados obtenidos.

Operación sobre datos (f_d)	Operación sobre residuos (f_r). Operación	Corrección
Suma	Suma	$-C_n\|m\|_A$
Negación	Negación	$\|m\|_A$
Diferencia	Diferencia	$(1-C_n)\|m\|_A$
Desplazamiento a la izquierda	Rotación a la izquierda	$\|X_{-1}-X_{n-1}\|2^n\|_A\|_A$
Desplazamiento a la derecha	rotación a la derecha	$\|X_n\|2^{n-1}\|_A-X_0 2^{a-1}\|_A$
"Y" lógico	Suma	$\|V(X_1 \vee X_2)\|_A$
"O" lógico	Suma	$\|V(X_1 \cdot X_2)\|_A$
"O" exclusivo	Suma	$\|2V(X_1 \cdot X_2)\|_A$

Tabla 4.2

CAPITULO V.

ESTRUCTURAS OPTIMAS DE DETECCION Y CORRECCION.

5.1. Introducción.

En la práctica del diseño de un computador, se puede dividir éste en varios subsistemas, cuyo diseño se emprende por separado para cada uno de ellos (teniendo siempre en cuenta la interación de cada subsistema con los demás).

Si se desea que el computador que se va a diseñar tenga características de Tolerancia a Fallos, se deberá dotar de dicha característica a cada uno de sus subsistemas. De este modo, se empleará para cada uno de ellos la estrategia de Tolerancia a Fallos mas adecuada.

Entre los subsistemas que constituyen la unidad central de un computador (memoria, unidad de proceso y unidad de control), las técnicas basadas en códigos se adaptan especialmente bien a los subsistemas de memoria principal, memoria de control y unidad de proceso de datos.

Dado que este trabajo se centra en las técnicas de Tolerancia a Fallos basadas en códigos, en este capítulo de orientación práctica se generan los códigos mas apropiados para proteger los subsistemas antes citados, planteando las estrategias óptimas de detección y corrección de errores y dando las bases para su implantación física en dichos subsistemas. Para ello nos basamos en los teoremas propuestos en el capítulo anterior.

5.2. Estrategias de corrección de error para memorias.

La memoria de un computador es históricamente el subsistema menos fiable de éste. Al mismo tiempo es el mas susceptible de aplicación de téc-

nicas de Tolerancia a Fallos. Esto es debido a su gran regularidad, a la sencillez de sus funciones y al gran número de elementos lógicos que la componen.

A lo largo de la historia de los computadores, se han utilizado para proteger su memoria técnicas de redundancia modular, técnicas de detección de error y conmutación de elementos en espera. Pero las técnicas que mejores resultados han dado son las de utilización de códigos detectores y correctores de error. Utilizando esta técnica se logra una mejor protección de la memoria con menos redundancia que con cuanquier otra.

En la elección del código para proteger la memória, lo que mas influye es la propia organización de ésta. Segun tengamos una organización u otra, habrá una estratégia óptima distinta. Vimos en el capítulo 3 que existen códigos con potencias de corrección muy diversas, pero en la memoria de un computador, los códigos óptimos son los correctores de un solo error o bien los correctores de un solo error y detectores de dos. Las razones principales son las siguientes:

- La complejidad y lentitud del decodificador se incrementa rápidamente a medida que aumenta el número de errores a corregir.

- Podemos organizar nuestra memoria de modo que los errores mas probables sean los simples con mucha diferencia sobre todos los demás.

Distinguiremos en un computador dos subsistemas de memoria: La memoria principal, caracterizada por contar con un elevado número de palabras de un ancho relativamente pequeño, y la memoria de control, que consta de relativamente pocas palabras de un ancho muy grande. Como vimos en el capítulo 2, en el mercado existen circuitos integrados de memoria con dos tipos de organización, una en rodajas de un bit y otra en rodajas de b bits (b = 4,8,...). Lógicamente, para la memoria principal es mas comun elegir los primeros y para la memoria de control los segundos.

5.2.1. Memoria principal.

Si organizamos nuestra memoria principal de modo que cada circuito integrado (o conjunto de circuitos integrados) corresponda a un solo bit de la palabra de memoria, podemos asegurar que un fallo en un solo circuito integrado afectará a un solo bit en una o mas palabras, y si se altera mas de una palabra como consecuencia de este fallo, la posición del error en cada una de ellas será la misma.

Como el hecho de que se produzca un fallo en un circuito integrado podemos considerarlo independiente de que se produzca fallo en cualquier otro, los errores en cada uno de los bits de las palabras de memoria serán independientes.

Sea P_i la probabilidad de que se produzca un error simple (en el bit i). La probabilidad de que se produzca un error doble (error en el bit i y en el bit j) será:

$$P_d = P_i P_j$$

Veamos algunos valores típicos: La probabilidad de que se produzca un fallo en un circuito integrado de memoria al cabo de 1000 horas de funcionamiento es del orden de 10^{-4}, es decir, $P_i = P_j = 10^{-4}$ (Levine 1976)

Así pues, la probabilidad de que se produzca un error doble es de $P_d = 10^{-8}$, que es cuatro ordenes de magnitud menor que la de error simple. La probabilidad de que se produzca un error múltiple (en mas de dos bits de la palabra de memoria) es por lo tanto despreciable.

Vemos entonces que, organizando la memoria principal en rodajas de un bit, para protegerla de un modo muy potente, bastará con utilizar un código de tipo paridad que sea capaz de corregir cualquier error simple. Para dar una protección adicional se puede utilizar un código que además sea capaz de detectar todos los errores dobles.

5.2.1.1. Elección del código.

En el capítulo 3 observamos que los códigos mas apropiados para su aplicación en computadores son los de bloques, y dentro de estos, los mas fáciles de codificar y decodificar son los lineales. Para distancias mínimas pequeñas (d = 3 y d = 4) hemos visto que los mas cómodos son los de Hamming. Este tipo de códigos permite ejecutar una detección y corrección simultánea, sin introducir grandes retardos en el funcionamiento libre de fallos.

Para la memoria hemos concluido en los párrafos anteriores que es suficiente un código capaz de corregir todos los errores simples (distancia mínima 3) o bien capaz de corregir todos los errores simples y detectar todos los dobles (distancia mínima 4). Entonces, podemos utilizar un código de Hamming o uno de Hamming ampliado respectivamente.

Aunque tanto la codificación como la decodificación con estos códigos es rápida, la necesidad de detección y corrección simultánea con el funcionamiento de la memoria nos lleva a tratar de minimizar los retardos introducidos por estos procesos. Por lo tanto, desarrollamos y formalizamos aquí los códigos de "peso impar mínimo" (de distancia mínima 4) debidos a Hsiao (Hsiao 1970), que permiten construir el codificador y el decodificador con un mínimo de elementos físicos, así como minimizar los retardos introducidos por éstos. Obtendremos códigos capaces de corregir todos los errores simples y detectar todos los errores dobles. Para ello nos basamos en el corolario 3.1 del capítulo 3 debido a Peterson (Peterson 1961):

"Un código lineal con matriz de paridad H tiene distancia mínima d si y solo si cualquier combinación de d-1 o menos columnas de la matriz H es linealmente independiente".

La capacidad de detección y corrección de que queremos dotar al código, nos obliga a buscar uno con distancia mínima 4. Por tanto, cualquier combinación de tres o menos columnas de la matriz H ha de ser linealmente independiente. Podemos satisfacer esta condición obligando a que las colum-

nas de H cumplan las siguientes restricciones:

1) Ninguna columna debe tener todos sus elementos nulos.

2) Todas las columnas deben ser distintas.

3) Todas las columnas deben contener un número impar de unos.

Las restricciones 1) y 2) obligan a que cualquier combinación de dos o menos columnas de H sea linealmente independiente (distancia mínima 3).

La restricción 3) obliga a que cualquier combinación de tres columnas de H sea linealmente independiente. Esto es así porque la suma módulo dos de tres vectores de peso impar es siempre un vector no nulo. En general lo es la suma módulo dos de un número impar de vectores de peso impar. De este modo, la distancia mínima de un código cuya matriz de paridad cumpla estas restricciones es 4.

La detección de errores dobles puede efectuarse teniendo en cuenta que la suma módulo dos de dos vectores de peso impar resulta un vector de peso par. Como en general la suma módulo dos de un número par de vectores de peso impar resulta siempre un vector de peso par, podemos también detectar todos los errores múltiples de peso par, tratándolos como dobles.

Basandonos en estas tres restricciones, obtenemos un procedimiento para construir la matriz H. Si tenemos k bits de información, necesitaremos r bits de paridad (redundantes). El procedimiento es el siguiente:

a) Los $\binom{r}{1}$ posibles vectores de peso 1 se usan como columnas para las r posiciones de los bits de paridad.

b) Si es $\binom{r}{3} \geq k$, tomamos k columnas de peso 3 entre los $\binom{r}{3}$ posibles vectores de peso 3. Si es $\binom{r}{3} < k$, tomamos los $\binom{r}{3}$ vectores, y el resto se seleccionan entre los $\binom{r}{5}$ vectores de peso 5, etc. El proceso se de-

tiene cuando se han seleccionado las k columnas necesarias.

Este procedimiento de construcción nos asegura que el número total de unos en la matriz es mínimo. Esto significa que el número de niveles lógicos necesarios tanto para codificar cómo para calcular el síndrome es mínimo como vemos a continuación.

Sea t_i el número de unos en la fila i de la matriz H. Sean C_i y S_i respectivamente el bit de paridad y el bit de síndrome correspondientes a la fila i de la matriz. Definimos:

l_{C_i} = Número de niveles lógicos necesario para generar C_i usando sumadores módulo 2 de v entradas.

l_{S_i} = Número de niveles lógicos necesario para generar S_i usando sumadores módulo 2 de v entradas.

Entonces, se verifica que:

$$l_{C_i} = \lceil \log_v(t_i - 1) \rceil$$

$$l_{S_i} = \lceil \log_v(t_i) \rceil$$

siendo $\lceil X \rceil$ el menor entero mayor o igual que X.

Como v queda fijado en la práctica por la familia de circuitos lógicos utilizados en la implantación física, el número de niveles lógicos será mínimo (y por tanto será mínimo el tiempo de decodificación y de generación de síndrome) cuando todos los t_i sean mínimos e iguales.

En general, este tipo de código, por tener número mínimo de unos en su matriz H, necesita menos elementos físicos para su realización que sus homólogos de Hamming, y menos elementos físicos significa tanto menor costo como mayor fiabilidad.

Comprobaremos ahora como estos códigos, a pesar de tener mayor capacidad de detección que los de Hamming e implicar menor costo y mayor fiabilidad, no necesitan un número mayor de bits de paridad que éstos.

En este tipo de códigos debe verificarse que:

$$\sum_{i=I}^{\leq r} \binom{r}{i} \geq r + k$$

Para un código de Hamming con distancia mínima 4 se verifica que:

$$2^{r-1} = k + r$$

Pero como es:

$$\sum_{i=0}^{r} \binom{r}{i} = 2^r \; y$$

$$\sum_{\substack{i=1 \\ i=impar}}^{\leq r} \binom{r}{i} = \sum_{\substack{i=0 \\ i=par}}^{\leq r} \binom{r}{i}, \text{ entonces:}$$

$$\sum_{\substack{i=1 \\ i=impar}}^{\leq r} \binom{r}{i} = \frac{1}{2} 2^r = 2^{r-1}$$

Vemos entonces que ambos códigos necesitan el mismo número r de bits de paridad.

En la tabla 5.1 se listan algunos códigos que pueden construirse mediante este procedimiento.

Si nos fijamos en la tabla, vemos por ejemplo el caso de n = 22, k = 16 y r = 6. En este caso, el número total de unos en la matriz H es 54, y el número medio 9, siendo el número de niveles lógicos $\lfloor \log_v 9 \rfloor$, frente a su correspondiente código de Hamming, para el cual el número total de unos sería 65, el número medio 10,8 y el número de niveles lógicos para el cálculo de uno de los bits del síndrome o de paridad sería $\lfloor \log_v 22 \rfloor$.

n	k	r	Estruc. de H $(\binom{r}{1},\binom{r}{3},\binom{r}{3})$	n.º total de 1's en H	n. medio de 1's en H	$\rceil s_i$
8	4	4	$\binom{4}{1}+\binom{4}{3}$	16	4	$\lceil \log_v 4 \rceil$
13	8	5	$\binom{5}{1}+8/\binom{5}{3}$	29	5,8	$\lceil \log_v 6 \rceil$
14	9	5	$\binom{5}{1}+9/\binom{5}{3}$	32	6,4	$\lceil \log_v 7 \rceil$
15	10	5	$\binom{5}{1}+\binom{5}{3}$	35	7	$\lceil \log_v 7 \rceil$
16	11	5	$\binom{5}{1}+\binom{5}{3}+\binom{5}{5}$	40	8	$\lceil \log_v 8 \rceil$
22	16	6	$\binom{6}{1}+16/\binom{6}{3}$	54	9	$\lceil \log_v 9 \rceil$
26	20	6	$\binom{6}{1}+\binom{6}{3}$	66	11	$\lceil \log_v 11 \rceil$
30	24	6	$\binom{6}{1}+\binom{6}{3}+4/\binom{6}{5}$	86	14,3	$\lceil \log_v 15 \rceil$
39	32	7	$\binom{7}{1}+32/\binom{7}{3}$	103	14,7	$\lceil \log_v 15 \rceil$
43	36	7	$\binom{7}{1}+\binom{7}{3}+1/\binom{7}{5}$	117	16,7	$\lceil \log_v 17 \rceil$
47	40	7	$\binom{7}{1}+\binom{7}{3}+9/\binom{7}{5}$	157	22,4	$\lceil \log_v 23 \rceil$

Tabla 5.1

5.2.1.2. Estructura de una memoria con corrección de errores.

En la figura 5.1 se muestra una estructura genérica de memoria con corrección de errores.

Su funcionamiento es el siguiente:

Escritura. El dato u_i de k bits que queremos almacenar, pasa por el codificador, donde se generan y se añaden los bits de paridad para formar la palabra código X_i (de n bits), que se almacena en la matriz de memoria.

Lectura. Cuando queremos leer un dato, a la salida de la matriz de

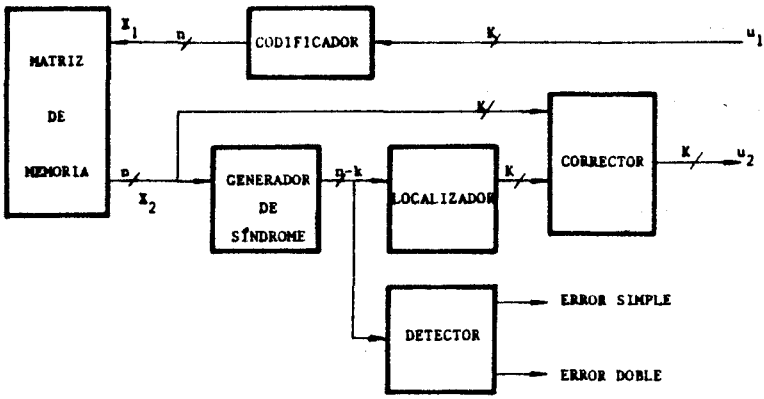


Figura 5.1

memoria aparece la palabra X_2, que va por una parte al generador de síndrome y por otra al corrector de error. El generador de síndrome obtiene un vector S de n - k componentes que contiene información sobre si se ha producido o no error, y en caso afirmativo si éste es o no corregible (simple o múltiple). Este síndrome S va al detector y al localizador, que establece en que posición de la palabra se ha producido el error. Por último, la salida del localizador se lleva al corrector, cuya salida será el dato correcto que queríamos leer.

A continuación veremos detalladamente con un ejemplo como se implantaría cada uno de estos bloques.

Ejemplo. Supongamos que queremos dotar de capacidad de corrección de errores simples y detección de errores dobles a una memoria con ancho de cuatro bits. Utilizamos para ello (por ser k = 4) el primer código listado en la tabla 5.1, que tiene n = 8, k = 4 y r = 4.

La matriz H de este código (obtenida siguiendo el procedimiento de

construcción indicado) es la siguiente:

$$H = \begin{matrix} 1\,1\,1\,0\,1\,0\,0\,0 \\ 1\,1\,0\,1\,0\,1\,0\,0 \\ 1\,0\,1\,1\,0\,0\,1\,0 \\ 0\,1\,1\,1\,0\,0\,0\,1 \end{matrix}$$

De esta matriz H obtenemos la correspondiente matriz generadora G:

$$G = \begin{matrix} 1\,0\,0\,0\,1\,1\,1\,0 \\ 0\,1\,0\,0\,1\,1\,0\,1 \\ 0\,0\,1\,0\,1\,0\,1\,1 \\ 0\,0\,0\,1\,0\,1\,1\,1 \end{matrix}$$

Construiremos ahora cada uno de los bloques del diagrama general.

-Codificador. La palabra código X se obtiene multiplicando el dato a almacenar u por la matriz G.

$X = uG = (u_1, u_2, u_3, u_4, u_1 \oplus u_2 \oplus u_3, u_1 \oplus u_2 \oplus u_4, u_1 \oplus u_3 \oplus u_4, u_2 \oplus u_3 \oplus u_4)$ Por tanto resulta el codificador de la figura 5.2

- Generador de síndrome. El síndrome se genera de un modo parecido:

$$S = HX^t = \begin{matrix} X_1 \oplus X_2 \oplus X_3 \oplus X_4 \\ X_1 \oplus X_2 \oplus X_4 \oplus X_6 \\ X_1 \oplus X_3 \oplus X_4 \oplus X_7 \\ X_2 \oplus X_3 \oplus X_4 \oplus X_8 \end{matrix}$$

De este modo queda el generador de síndrome de la figura 5.3.

- Localizador. Este módulo debe identificar cada síndrome con una columna de la matriz H, lo que dará la posición del bit erróneo. Se implantaría como muestra la figura 5.4

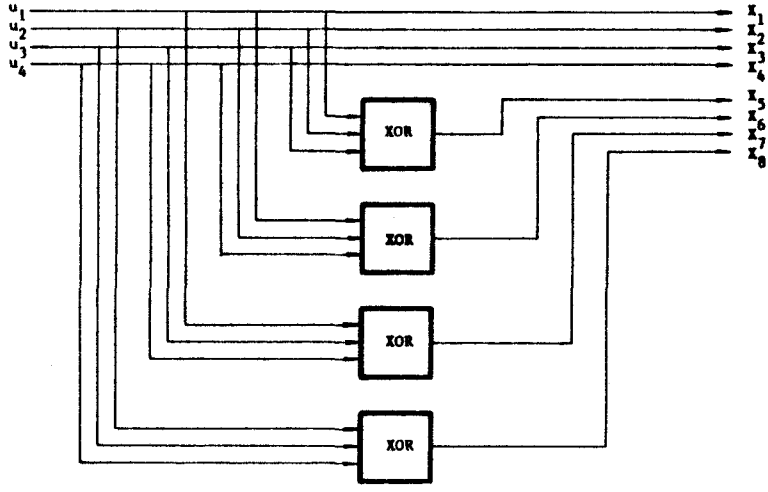


Figura 5.2

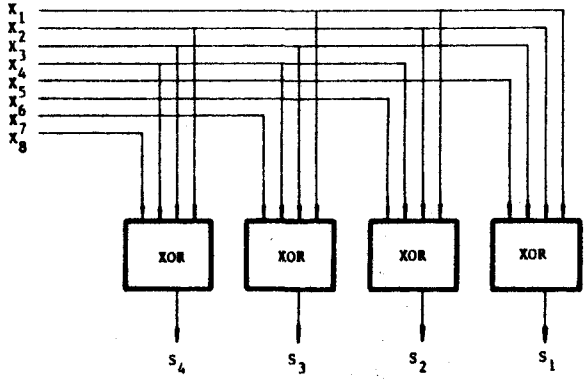


Figura 5.3

- Corrector de error. Este módulo debe únicamente invertir el bit indicado por el localizador. Su implantación se muestra en la figura 5.5.

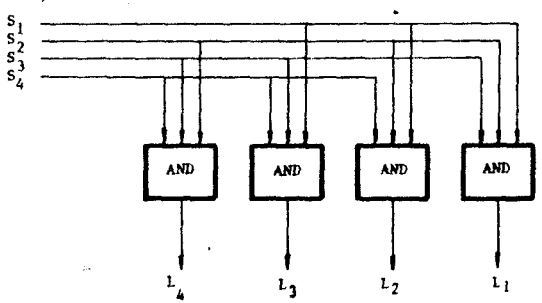


Figura 5.4

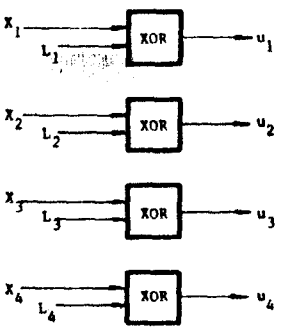


Figura 5.5

- Detector de error. Su única misión es determinar si el síndrome es cero, en cuyo caso indica que no se ha producido error. Si no es cero, distingue si se ha producido error en un solo bit o en varios verificando la paridad del síndrome. Se muestra en la figura 5.6

Con todo lo anterior hemos establecido las bases de la implantación física de un sistema de detección y corrección simultánea de errores, uti-

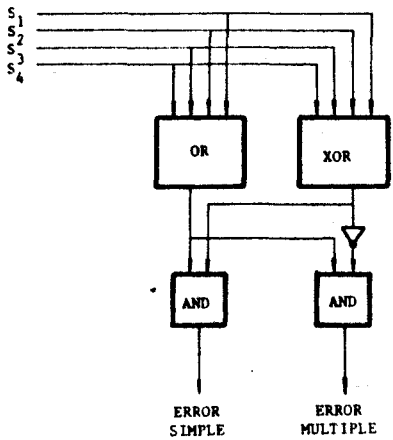


Figura 5.6

lizando para ello códigos de peso impar mínimo capaces de corregir cualquier error simple y detectar cualquier error múltiple de peso par.

Esta protección es suficiente para aplicaciones normales. Si necesitamos una memoria con una fiabilidad tan extremadamente grande que no es suficiente con esta potencia de corrección, no queda mas remedio que pasar a utilizar códigos mas potentes, tales como los BCH (correctores de error doble), o bien hacer una sofisticación del decodificador como veremos a continuación, que nos permitirá (basandonos en el tipo de error que se produce en los circuitos integrados de memoria) corregir errores dobles sin necesidad de utilizar la gran redundancia de los códigos BCH.

5.2.1.3. Corrección de "borrados".

En casos muy especiales en que necesitemos una fiabilidad muy extremada, podemos utilizar como ya hemos dicho un código BCH con distancia mínima 5, es decir, capaz de corregir todos los errores dobles. La utilización de este código supone un gran incremento de la complejidad del codifi-

cador y decodificador así como un gran aumento del número de bits de paridad. Por ejemplo, para datos de 16 bits, con un código con distancia mínima 4 se necesitan 6 bits de paridad (22,16). Para utilizar un código BCH con distancia 5 se necesitarían 10 bits de paridad (26,16).

La segunda alternativa es sofisticar el decodificador y utilizar un código con distancia 4 para corregir un error simple y un borrado. Sundberg (Sundberg 1978) y Walker et al. (Walker 1979) desarrollaron un método de decodificación que utiliza este concepto de borrado para el caso en que los errores que se produzcan en la memoria sean determinados ("pegado a"). A continuación desarrollamos este mismo método haciendolo válido para cualquier tipo de error que se produzca en los circuitos integrados de memoria.

Al contrario que en los canales de transmisión, donde los errores se producen aleatoriamente en cualquier posición de la palabra código, en una memoria organizada en rodajas de un bit (uno o varios circuitos integrados por cada bit de la palabra de memoria), si se produce un fallo en un circuito integrado, éste induce un error en una o varias palabras del código, pero con la característica de que el bit erróneo (cuando lo haya) siempre estará en la misma posición de la palabra, ya sea un fallo de tipo determinado o de tipo indeterminado. Además, lo mas probable es que si falla mas de un circuito integrado, estos fallos se produzcan de un modo escalonado en el tiempo, lo que permitirá tener en el decodificador información sobre el primer error cuando se presenten los efectos combinados del primero y el segundo.

Por todo esto, podemos utilizar en el decodificador el concepto de "borrado" (Peterson 1961). Como borrado se define un símbolo de una palabra código en una posición conocida, pero de valor desconocido, es decir, potencialmente erróneo. Es mucho mas facil corregir un borrado que un error.

Sabemos por teoria de códigos (Peterson 1961), que un código con distancia mínima d puede corregir todas las combinaciones de errores aleatorios de peso t y borrados de peso r si se verifica que:

$$2t + r < d,$$

siendo d su distancia mínima.

De este modo, podemos utilizar un código con distancia mínima 4 para corregir errores simples (en un bit) y borrados también en un bit. Veamos la estructura y funcionamiento del decodificador. La estructura de la memoria será la mostrada en la figura 5.7.

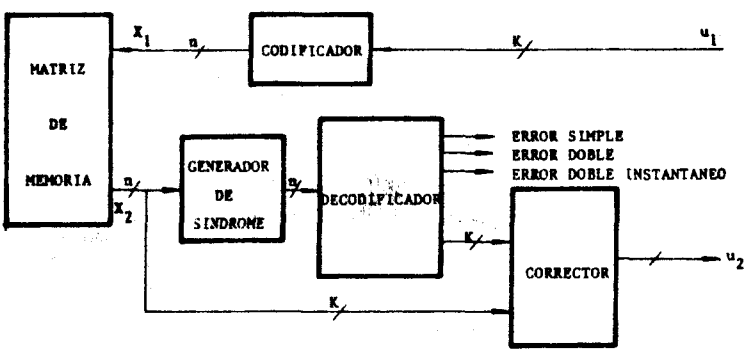


Figura 5.7

El decodificador se encarga tanto de generar las señales de detección de error como de generar el valor del error (localizarlo) para efectuar la corrección. La estructura interna del decodificador se muestra en la figura 5.8.

Su funcionamiento es el siguiente:

Cuando se detecta por primera vez un error simple en la memoria, se almacena el síndrome S_a en el registro correspondiente. Al mismo tiempo, se corrige el error como tal error simple. Cuando algún tiempo después se detecta un error doble, se utiliza para su corrección el síndrome almacenado

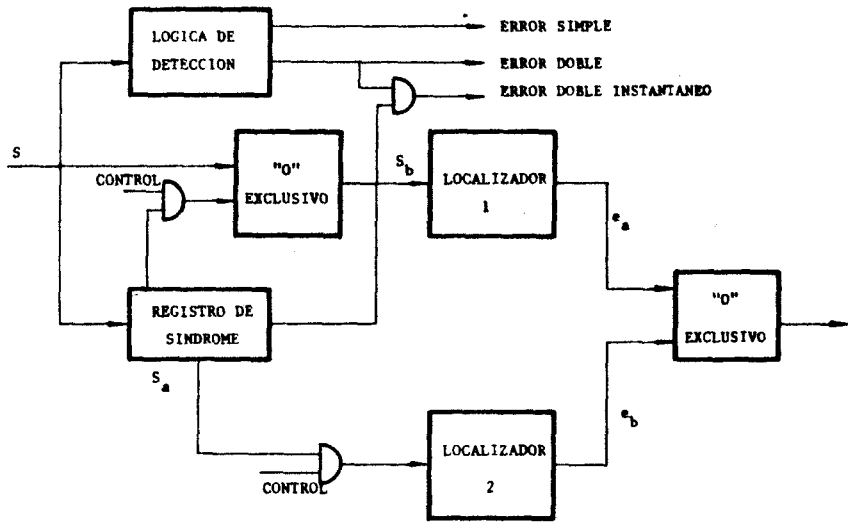


Figura 5.8

S_a. Dicha corrección se efectua como sigue:

a) Se detecta el error doble.

b) El síndrome recibido S corresponde a un error doble (es lógico que corresponda al error cuyo síndrome está almacenado, junto con un nuevo error). Calculamos $S_b = S \oplus S_a$, que corresponde al síndrome del segundo error simple.

c) Corregimos el dato con la secuencia de error $e = e_a \oplus e_b$.

Cuando después de tener un síndrome almacenado S_a, el error que se detecta es simple, no se utiliza S_a para la corrección. Este error simple puede ser el mismo que el primero, o bien otro distinto. Esto puede suceder cuando la palabra leida no queda afectada por el fallo (está fuera de la

zona dañada del circuito integrado) o bien cuando el fallo es del tipo determinado y el valor almacenado corresponde al valor constante del bit.

Mediante esta estrategia podemos corregir prácticamente todos los errores dobles que se produzcan en una memoria. El único caso que no se puede corregir automáticamente es cuando se produce un error doble instantáneo, es decir, en el caso muy poco probable de que fallen dos circuitos integrados en el mismo ciclo de memoria. En este caso, como no tenemos almacenado el síndrome S_a, no se puede ejecutar la corrección. De todos modos, este caso es detectable, y puede lanzarse una rutina de localización del fallo.

Concluimos entonces que con esta estrategia de corrección, con un código con distancia 4 (que teoricamente solo puede corregir errores simples y detectar errores dobles), sofisticando un poco el decodificador alcanzamos practicamente la misma capacidad de corrección que tendríamos usando un código BCH con distancia 5 (capaz de corregir errores dobles), pero sin necesidad de aumentar el número de bits de paridad, es decir, la redundancia.

5.2.2. Memoria de control.

La memoria de control de un computador suele contar con un numero relativamente bajo de palabras, pero de un ancho considerable. Un caso típico podría ser 1 K-palabras de 64 bits. A la hora de organizar esta memoria vemos que, para aprovechar la capacidad de integración de la tecnología de semiconductores, no debemos hacerlo en rodajas de un bit, sino utilizar la otra estructura de circuitos integrados de memoria, y dividir ésta en rodajas de b bits, donde cada rodaja corresponda a un solo circuito integrado.

El inconveniente de está organización es que si se produce un fallo en un circuito integrado de la matriz de memoria, como el error a su salida suele ser distribuido, en la palabra de memoria puede haber mas de un bit erróneo, por lo cual no podemos utilizar para su protección los códigos co-

rrectores de error simple que utilizamos en la memoria principal, puesto que la mayoría no van a ser errores simples.

Por esta razón, para proteger la memoria de control debemos utilizar códigos que sean capaces de corregir errores en rodajas de b bits. En el capítulo 3 se introdujeron los códigos b-adyacentes debidos a Bossen (Bossen 1970). Aqui desarrollamos un tipo específico de estos códigos, los llamados "birredundantes" que son fáciles de codificar y decodificar.

5.2.2.1. Códigos birredundantes.

Estos códigos constituyen una subclase de los códigos de tipo Hamming sobre el campo finito $GF(2^b)$. Tienen dos símbolos redundantes y son capaces de corregir todos los errores simples en $GF(2^b)$, es decir, que corrigen cualquier error que se produzca en un grupo de b bits. Su matriz de paridad H es de la forma:

$$H = \begin{matrix} 1\ 1\ 1\ 1\ \ldots\ 1\ 1\ 0 \\ 1\ B_1 B_2 \ldots B_{k-1}\ 0\ 1 \end{matrix}$$

Donde $B_1, B_2, \ldots, B_{k-1}$ son elementos de $GF(2^b)$. Por ser estos B_i distintos, no existe ninguna combinación de dos columnas de H que sea linealmente dependiente, y por lo tanto, la distancia mínima del código es 3. Así, con este código se puede corregir cualquier error simple (en dígitos de b bits). El máximo número de dígitos de información que se puede codificar es igual al número de elementos no nulos y distintos en $GF(2^b)$, que es $2^b - 1$.

Tomando como polinomio primitivo el polinomio P(x) que define $GF(2^b)$, todas las potencias de $a = (x)$ serán distintas, puesto que cada B_i es igual a una potencia de a.

De este modo, la matriz de paridad H queda:

$$H = \begin{matrix} 1 & 1 & 1 & \ldots & 1 & 1 & 0 \\ a^0 & a^1 & a^2 & \ldots & a^{2^b-2} & 0 & 1 \end{matrix}$$

en la que basta sustituir cada a^i por su correspondiente matriz de transformación T_{a^i} para obtener la matriz de paridad H en forma binaria. La ventaja fundamental de estos códigos, aparte de su baja redundancia, es la facilidad con que pueden decodificarse.

Supongamos que se produce un error en b o menos bits de la rodaja i. Este error corresponde a algun $e_i \in GF(2^b)$. El síndrome correspondiente es:

$S = \begin{pmatrix} S_1 \\ S_2 \end{pmatrix} = \begin{pmatrix} e_i \\ a^i e_i \end{pmatrix}$ donde S_1 y S_2 son vectores binarios de b componentes. Si el error ha ocurrido en alguna de las dos rodajas redundantes, el síndrome toma el valor:

$$S = \begin{pmatrix} S_1 \\ S_2 \end{pmatrix} = \begin{pmatrix} e_i \\ 0 \end{pmatrix}, \text{ o bien}$$

$$S = \begin{pmatrix} 0 \\ e_i \end{pmatrix}$$

dependiendo de que el error esté en el primero o en el segundo dígito redundante. Podemos detectar fácilmente ambos casos mediante puertas "Y" lógico que comprueben si es $S_1 = 0$ o $S_2 = 0$. Esto es posible porque si no hay error en la parte redundante, e_1 y e_2 no pueden anularse a la vez.

Si $e \neq 0$, es $a^i e \neq 0$ para todo $a^i \in GF(2^b)$; $a \neq 0$

Para un síndrome S con sus dos componentes distintos de cero, el error está en el bloque i si y solo si se verifica que:

$$a^i S_1 = S_2$$

En cuyo caso, el valor del error $e_i = S_1$ debe sumarse módulo 2 al bloque i para corregir dicho error. La estructura del decodificador sería la mostra-

da en la figura 5.9.

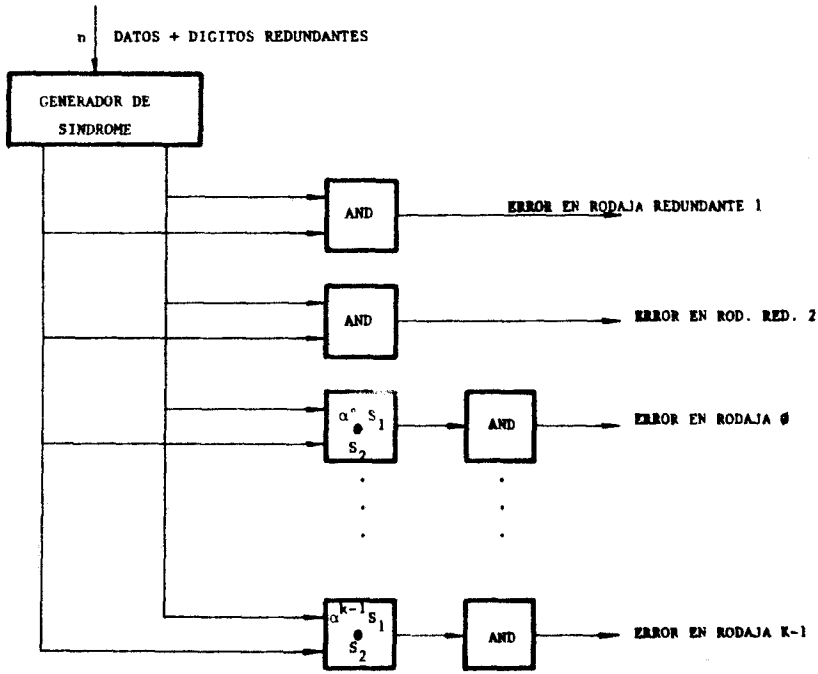


Figura 5.9

Este método de decodificación necesita un número mucho menor de elementos físicos que una decodificación directa del síndrome. Vemos entonces que, utilizando un código birredundante podemos corregir cualquier error que se produzca en una rodaja de b bits en la memoria.

Un ejemplo de código birredundante es el código (80,64) para rodajas con b = 8 bits. Usando este código necesitaríamos 10 circuitos integrados de memoria (8 para datos y dos redundantes). Así podríamos corregir cualquier error debido al fallo de uno de los circuitos integrados de memo-

ria.

5.3. Unidad de Proceso.

Planteamos y desarrollamos en este apartado estrategias para proteger la unidad de proceso de un computador. Se basan en dividir la unidad verticalmente en rodajas de un ancho determinado. La primera utiliza para la protección un código aritmético corrector de errores, la segunda utiliza una mezcla de duplicación funcional con un código corrector.

La división en rodajas está motivada por las siguientes razones:

- La partición en rodajas va generalmente unida a las técnicas de anticipación de acarreo, tanto entre rodajas como en el interior de éstas. Esto permite obtener una velocidad de proceso mayor.

- El estado actual de la tecnología de semiconductores permite incluir la rodaja completa en un solo circuito integrado.

- Como contrapartida, se presenta la dificultad de que, como vimos en el capítulo 2, los fallos que se producen en circuitos construidos con integración a gran escala suelen dar lugar a errores distribuidos, es decir, que pueden afectar a mas de una salida de la rodaja. Por esta razón, los códigos aritméticos correctores de un solo error pueden resultar muy poco efectivos.

- Pero, como veremos a continuación, podemos obtener códigos aritméticos con baja redundancia que son capaces de corregir cualquier error que se presente en una sola rodaja.

5.3.1. Código corrector de error en una rodaja.

Planteamos primero una estrategia en la que el único método de protección es la aplicación de un código corrector de error. El primer problema es establecer que tipo de código aritmético es el mas apropiado, y ade-

más definir un código concreto que tenga capacidad para corregir errores en la rodaja definida.

5.3.1.1. Determinación del tipo de código mas apropiado.

Determinamos cual es el tipo de código aritmético mas apropiado haciendo un estudio comparativo de los códigos aritméticos correctores de error mas comunes. Estos son: Códigos AN (no separados), códigos gAN (sistemáticos no separados) y códigos multirresiduo (separados). Las características diferenciales de estos códigos son las siguientes:

- Capacidad de representación y razón de redundancia.

Vimos en el capitulo 3 que, tanto con los códigos AN como con los gAN, la palabra código mas grande que es posible representar para una capacidad de corrección dada, queda determinada por la cantidad $AM_r(A,d)$, es decir, que el dato mas grande que podemos codificar es el entero $M_r(A,d)$.

Sabemos por la correspondencia entre códigos AN y códigos multirresiduo, que el máximo dato representable con estos últimos es precisamente $AM_r(A,d)$, lo que significa por un lado un incremento en el rango de representación, y por otro una disminución de la razón de redundancia para la misma capacidad de corrección. Veamos esto con un ejemplo.

Tomemos dos códigos equivalentes en capacidad de corrección, como son el código AN con A = 15 x 511 y el código birresiduo con bases $m_1 = 15$ y $m_2 = 511$ que, como veremos mas adelante, son capaces de corregir cualquier error en una rodaja de cuatro bits. Estos códigos tienen distancia mínima 3 y su rango de representación es:

$$AM_r(A,d) = 16^9 - 1 = 2^{36} - 1.$$

Si trabajamos con el código AN, lo anterior quiere decir que el tamaño máximo de la palabra código es de 36 bits, en los cuales están comprendidos 13 bits redundantes. Por lo tanto, solo podemos codificar datos

que tengan un ancho de hasta 23 bits. La palabra código (para un código gAN) tendría la estructura de la figura 5.10.

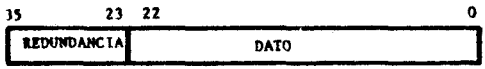


Figura 5.10

Para ambos AN y gAN la razón de redundancia definida como el cociente entre el número de bits redundantes y el número total de bits sería la siguiente:

$$R = \frac{r}{n} \qquad R = \frac{13}{36} = 0,36$$

En cambio, si trabajamos con el código birresiduo de bases $m_1 = 15$ y $m_2 = 511$, el dato máximo que se puede codificar tiene un tamaño de 36 bits. La palabra código tiene la estructura de la figura 5.11.

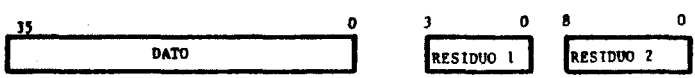


Figura 5.11

La razón de redundancia sería:

$$R = \frac{13}{49} = 0,26$$

Como vemos, también la razón de redundancia es mucho mejor para los códigos birresiduo que para los AN y los gAN.

- Facilidad de codificación y decodificación.

En general, los códigos AN y gAN necesitan para la codificación una unidad que multiplique el dato a codificar por la cantidad A. Para el cálculo del síndrome, lo que necesitan es otra unidad que calcule el residuo módulo A de la palabra código (división).

Los códigos multirresiduo, tanto para codificar como para el cálculo del síndrome necesitan una unidad que calcule el residuo de un número módulo cada una de las bases.

La máxima simplificación que puede hacerse es utilizar códigos de bajo costo, lo que significa tomar A para los códigos AN de la forma $A = 2^a - 1$, y para códigos multirresiduo, cada una de las bases m_i de la forma $m_i = 2^{t_i} - 1$. Hacer esto en códigos AN y gAN únicamente simplifica el cálculo del síndrome, pues se sigue necesitando un multiplicador. En cambio, con los códigos multirresiduo se simplifican ambos procesos, ganando así tanto en sencillez de implantación física como en velocidad de operación.

- Operaciones no aritméticas.

Para cierto tipo de operaciones, como los desplazamientos y las operaciones lógicas, los códigos AN y gAN no permanecen cerrados. Es mas, por ser códigos no separados, si queremos ejecutar por ejemplo una operación lógica, debemos primero decodificar las palabras código, ejecutar la operación, y a continuación recodificar el resultado. Este proceso, además de ser lento y complicado, provoca que este tipo de operaciones no queden protegidas por el código.

En cambio, como los códigos multirresiduo son separados, para ejecutar una operación de este tipo no es necesario decodificar las palabras código. Además, para que la operación quede protegida por el código, basta con aplicar los resultados obtenidos en el capítulo 4 de este trabajo.

Vemos entonces que, en general, los códigos multirresiduo tienen grandes ventajas frente a todos los demás aritméticos. Por tanto, son los que utilizamos en el desarrollo posterior.

5.3.1.2. Generación del código.

Una vez determinado que el tipo de código mas apropiado para proteger una unidad de proceso es un multirresiduo, el problema es definir uno que sea capaz de corregir todos los errores que se produzcan en una rodaja de dicha unidad de proceso.

Vimos en el capítulo 3 que podemos utilizar los códigos birresiduo para corregir cualquier error que se produzca en un solo bit de la unidad de proceso, puesto que son códigos correctores de errores simples. Rao en (Rao 1974) desarrolla un ejemplo de código birresiduo que es capaz de corregir cualquier error simple en datos de 20 bits. La observación de este ejemplo, y en concreto el elevado número de posibles síndromes de error distintos ($2^9 - 1$), frente al pequeño número de ellos utilizado (32), nos indujo a pensar que el código tenia capacidad para corregir errores mas complejos. Así pues, tratamos de utilizar este mismo código para corregir errores en rodajas de 4 bits. Se hizo una caracterización de cada uno de los síndromes (mediante un programa de simulación), y se comprobó que todos los errores de la forma $\pm a16^j$ para $0 < a < 16$ tienen síndromes distintos. Se comprobó así que el código es capaz de corregir dichos errores.

En un intento de formalización y generalización de este resultado, después de una intensa búsqueda bibliográfica, dimos con el trabajo de Neumann y Rao (Neumann 1975) que apunta hacia este resultado. De aqui hemos obtenido los siguientes tres teoremas, cuyas demostraciones hemos adaptado y que nos permiten generar códigos para corrección de errores en rodajas de b bits.

Los resultados que vamos a obtener a continuación, debido a la correspondencia vista en el capítulo 3, son directamente aplicables tanto a los códigos AN y gAN, como a los birresiduo.

Sea una base de representación r. Un error simple en un número con dicha base de representación tiene la forma $\pm ar^j$, con $0 < a < r$. Demostraremos que este tipo de error es corregible mediante un código AN con generador A de la forma $A = (r - 1)p$ (y por lo tanto, por un código birresiduo con bases de la forma $m_1 = r - 1$ y $m_2 = p$), siendo p un número primo mayor que r, que debe satisfacer ciertas condiciones.

Trabajamos entonces con $\hat{n}$ dígitos en base r ($\hat{n}$ rodájas). Utilizando aritmética de complemento a la base o de complemento restringido a la base, un error simple E, se define como un error de peso modular uno, y viene determinado por:

$$E = ar^j \quad \text{o bien} \quad E = m - ar^j,$$

donde $\quad 0 < |a| < r$, y $0 \leq j < \hat{n}$

j representa la posición del dígito erróneo, y

$$m = r^{\hat{n}}, \text{ o bien } m = r^{\hat{n}} - 1.$$

Teorema 5.1.

Sea un código AN con $A = (r - 1)p$. *Para cualquier número primo* $p > r$ se verifica que para todo a tal que $0 < a < r - 1$

$$p - 1 \notin G_r(p), \text{ y } (a - r + 1)a^{-1} \notin G_r(p) \text{ si y solo si}$$

$$AM_r(A,d) = r^{e_r(p)} - 1$$

$G_r(p)$ denota el subgrupo multiplicativo (r (mod. p)), y $aa^{-1} \equiv 1 (\text{mod. } p)$

Demostración.

1) *Supongamos que* $p - 1 \in G_r(p)$. *Entonces, podemos poner que* $p - 1 \equiv r^i \pmod{p}$. Como $p - 1 \equiv -1 (\text{mod. } p)$, debe ser $i = e_r(p)/2$. Por lo

tanto:

$r^{e_r(p)/2} + 1 \equiv 0 \pmod{p}$, lo que implica que:

$(r - 1)(r^{e_r(p)/2} + 1) \equiv 0 \pmod{A}$

Como $(r - 1)(r^{e_r(p)/2} + 1)$ tiene peso dos, debe verificarse que:

$AM_r(A,3) < r^{e_r(p)} - 1$,

que es contradictorio, por lo tanto, debe ser $p - 1 \notin G_r(p)$.

2) Sea $p - 1 \notin G_r(p)$ y $(a - r+1)a^{-1} \notin G_r(p)$

Como $r^{e_r(p)} - 1 \equiv 0 \pmod{A}$, y $r^{e_r(p)} - 1$ tiene peso dos, debe ser:

$AM_r(A,3) \ll r^{e_r(p)} - 1$.

Consideremos el anillo R de los números enteros módulo $r^{e_r(p)} - 1$. En R, si cualquier error de peso uno tiene síndrome único respecto de A, cualquier palabra código distinta de cero debe tener peso mayor que dos. Por lo tanto, $AM_r(A,3)$ debe ser igual a $r^{e_r(p)} - 1$. Asi pues, basta con demostrar que para cualquier pareja de errores distintos $E_i = ar^j$ y $E = b^l$, con $0 < |a| < r$ y $0 < |b| < r$ para j y l pertenecientes al conjunto $(0, 1, \ldots, e_r(p) - 1)$, sus síndromes son distintos.

Supongamos que los síndromes son iguales:

$ar^j \equiv br^l \pmod{A}$. Por lo tanto, $ar^{j-1} \equiv b \pmod{A}$

Así, $ar^i = b \pmod{A}$ con $i = j - 1$. Por tanto, debe cumplirse:

$$ar^i \equiv b \pmod{r - 1} \quad (1)$$

$$ar^i \equiv b \pmod{p}$$

De (1) obtenemos que si a y b tienen el mismo signo, es a = b. Si a y b tienen signos opuestos, podemos tomar a > 0 (sin perder generalidad), con lo que se verifica:

$$a = b + (r - 1)$$

Cuando es a = b, no se satisface (2) para ningún i, salvo en el caso en que $E_1 = E_2$. Por lo tanto, los síndromes de todos los errores distintos son distintos.

Cuando es a ≠ b, tenemos:

$ar^i \equiv a - (r - 1) \pmod{p}$, así $(a - r + 1)a^{-1} \equiv r^i \pmod{p}$, lo que implica que:

$(a - r + 1)a^{-1} \in G_r(p)$, que contradice la hipótesis. c.q.d.

Mediante la aplicación de este teorema, podemos comprobar (no sin dificultad) si el rango de representación dado por $AM_r(A,3)$ para p y r dados es suficiente para nuestra aplicación. De todos modos, a continuación vemos que si restringimos los valores de p y r a un tipo particular, podemos facilmente calcular $AM_r(A,3)$ para r grande.

Lema 5.1.

Dado $r = 2^b$ y un primo $p = 2^d - 1$, si $p > r$ se verifica que $p - 1 \notin G_r(p)$ y $(a - r + 1)a^{-1} \notin G_r(p)$.

Demostración.

Como p es primo, d también debe serlo. Entonces, los elementos de $G_r(p)$ son precisamente las primeras d potencias consecutivas de 2, puesto que en este caso es $G_r(p)$ idéntico a $G_2(p)$, y el máximo comun divisor de b y d es 1. Por tanto, solamente debemos probar que para ningún l se verifica la congruencia:

$$a2^l \equiv a - r + 1 \pmod{2^d - 1}$$

Supongamos que se verifica. Entonces,

$$a2^l \equiv 2^d - 2^b + a \pmod{2^d - 1}, \text{ por tanto,}$$

$$a2^l \equiv 2^b(2^{d-b} - 1) + a \pmod{2^d - 1}$$

Vemos entonces que en el lado derecho de la congruencia hay un entero menor que $2^d - 1$, cuya representación binaria tiene dos partes, la de orden superior, de valor $2^d - 2^b$, y la de orden inferior (b dígitos) de valor a. También vemos que el peso de Hamming de este entero debe superar al menos en uno al peso de a. Por otra parte, el peso de Hamming de $a2^l \pmod{2^d - 1}$ es exactamente el peso de a. Así pues, la congruencia de partida no puede verificarse.

Este lema nos da pie para demostrar el siguiente teorema.

Teorema 5.2.

Para $r = 2^b$ y un primo $p = 2^d - 1$, con $p > r$, si tomamos $A = (r - 1)p$, se verifica que:

$$AM_r(A,3) = r^d - 1$$

Demostración.

Del lema 5.1, como $r = 2^b$, $p = 2^d - 1$ y $p > r$, se verifica que $(a - r + 1)a^{-1} \notin G_r(p)$. También, los elementos de $G_r(p)$ son de la forma 2^k, con $k = 0,1,\ldots,d - 1$, y $p - 1 \notin G_r(p)$. De este modo, como el máximo comun divisor de b y d es uno, se verifica que $e_r(2^d - 1) = d$. Entonces, aplicando el teorema 5.1, queda demostrado que:

$$AM_r(A,3) = r^d - 1 \quad \text{c.q.d.}$$

Este importante teorema nos proporciona el método para calcular el rango de representación $AM_r(A,3)$ ara r y p dados. Por ejemplo, si trabajamos con $r = 16$ (rodajas de cuatro bits), el menor valor de p de la forma $2^d - 1$ que podemos utilizar es $p = 31$. Es decir, que como $p = 2^5 - 1$ ($d = 5$), calculamos $AM_r(A,3)$ que es:

$$AM_r(A,3) = 16^5 - 1 = 2^{20} - 1$$

En lo anterior hemos supuesto que p es primo. Sin embargo, también podemos encontrar códigos correctores de error en una rodaja para muchos enteros no primos (usamos q en lugar de p para denotar que no necesita ser primo), con $A = (r - 1)q$.

Con el siguiente teorema generalizamos los resultados anteriores a los enteros no primos de la forma $q = 2^d - 1$. En estas condiciones, existen códigos para cualquier $d > 3$ (excepto 4 o 6) para al menos un $r = 2^b \geq 4$.

Teorema 5.3.

Sea $A = (r - 1)q$, con $r = 2^b$, $q = 2^d - 1$ $(d > b)$ y con $MCD(r - 1,q) = 1$. Sea f el mayor número entero (no necesariamente primo) para el cual $2^f - 1$ es un divisor de q. Entonces se verifica:

$$AM_r(A,3) = r^d - 1 \text{ si y solo si } r \leq q/(2^f - 1)$$

Demostración.

Si $r > q/(2^f - 1)$, entonces los dos errores $qr^f/(2^f - 1)$ y $qr^0/(2^f - 1)$ tienen síndromes iguales, con lo cual no se puede hacer la corrección del error. Esto se deduce simplemente de que $r^f - 1 \equiv 0 \pmod{r - 1}$ y $r^f - 1 = (2^f)^b - 1 \equiv 0 \pmod{2^f - 1}$. Así,

$$A = (r - 1)q = (r - 1)q(2^f - 1)/(2^f - 1)$$

que divide a la diferencia de los dos errores $q(r^f - 1)/(2^f - 1)$. Así pues,

estos dos errores no pueden tratarse como simples.

Como $r^i - 1$ es múltiplo de $2^d - 1$ solamente para $1 \leq i < d$, entonces todos los síndromes son distintos. c.q.d.

Un ejemplo de código obtenido por la aplicación de este teorema es el siguiente:

Tomemos $r = 16$, $q = 511$ ($r = 2^4$, $q = 2^9 - 1$)

Como $q = 511 = 7x73$, entonces, $f = 3$ y $r = 16 < q/(2^f - 1) = 73$

Por tanto, podemos calcular $AM_r(A,3)$ como $r^d - 1$. Así:

$$AM_r(A,3) = (2^4)^9 - 1 = 2^{36} - 1$$

En las tablas 5.2 y 5.3 listamos algunos códigos correctores de error en una rodaja para distintas bases r, indicando su rango de representación, así como el número de bits necesarios para ésta.

	r = 4		r = 8		r = 16	
p	$AM_r(A,3)$	n	$AM_r(A,3)$	n	$AM_r(A,3)$	n
31	$4^5 - 1$	10	$8^5 - 1$	15	$16^5 - 1$	20
127	$4^7 - 1$	14	$8^7 - 1$	21	$16^7 - 1$	28

Tabla 5.2. Obtenidos por aplicación del Th 5.2

Hemos demostrado unos teoremas mediante los cuales podemos calcular el rango de representación del código con el que vamos a trabajar. Como citábamos antes, estos resultados son aplicables tanto a los códigos AN y los gAN como a los códigos birresiduo.

	r = 4		r = 8		r = 16	
q	$AM_r(A,3)$	n	$AM_r(A,3)$	n	$AM_r(A,3)$	n
255	---	-	$8^8 - 1$	24	---	-
511	$4^9 - 1$	18	---	-	$16^9 - 1$	36
1023	---	-	$8^{10} - 1$	30	$16^{10} - 1$	40
2047	$4^{11} - 1$	22	$8^{11} - 1$	33	$16^{11} - 1$	44
$2^{17} - 1$	$4^{17} - 1$	34	$8^{17} - 1$	51	$16^{17} - 1$	68

Tabla 5.3. Obtenidos por aplicación del Th. 5.3

5.3.1.3 Estructura de la unidad de proceso.

Planteamos ahora de un modo general cómo debe construirse una unidad de proceso protegida contra errores mediante el uso de un código corrector birresiduo.

El procesador tiene como entradas dos datos X_1 y X_2 de longitud n bits, así como las señales de control que indican la operación f_d a ejecutar. Debe dar como salida el resultado correcto de la operación, y unas señales indicadoras de error (figura 5.12).

La unidad consta de los siguientes bloques fundamentales:

- Procesador de datos, capaz de ejecutar tanto operaciones lógicas como aritméticas sobre operandos de n bits. Estará dividido en rodajas del ancho que el código sea capaz de corregir.

- Procesadores de residuo. Ejecutan sobre los residuos de los operandos la operación f_r correspondiente a la f_d que se ejecuta sobre éstos.

- Codificadores. Son unidades que generan los residuos módulo $m_1 = 2^{c_1} - 1$ y módulo $m_2 = 2^{c_2} - 1$.

- Generador de síndrome. Este bloque se encarga de generar, par-

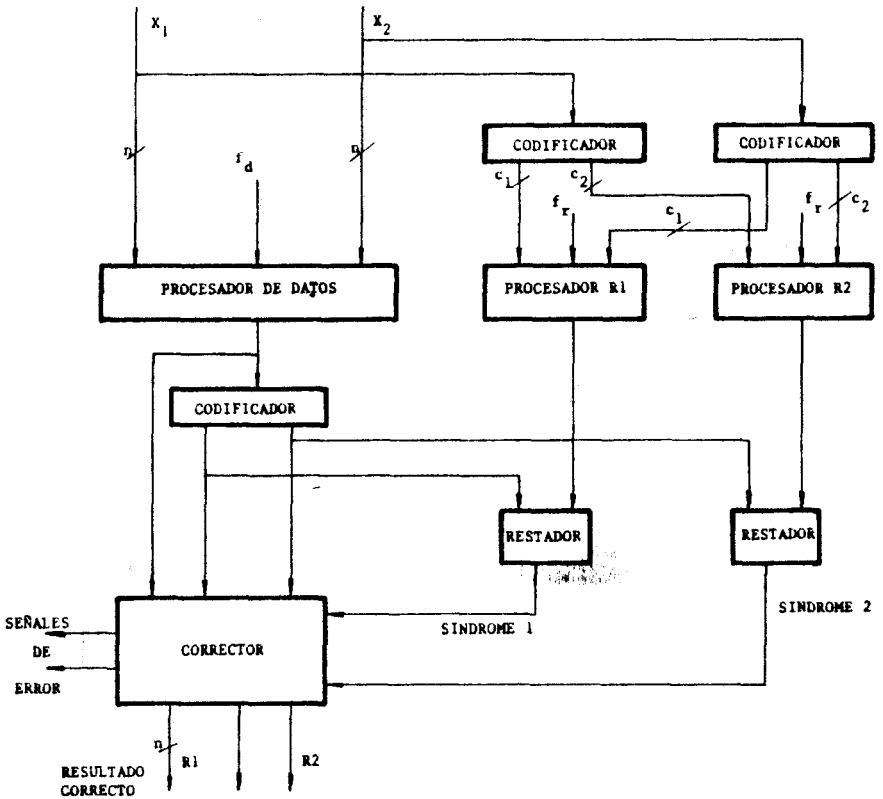


Figura 5.12

tiendo del resultado de operar sobre los datos y del resultado de operar sobre los residuos, el síndrome del error. Consta este bloque de un codificador que genera los residuos módulo m_1 y m_2 del resultado de operar sobre los datos, un restador módulo m_1 y otro módulo m_2 donde se restan los residuos para obtener el síndrome.

- Corrector de error. Se encarga de, partiendo del síndrome y del resultado de operar con los datos, generar el resultado correcto de la ope-

ración. También genera los residuos correctos y algunas señales de indicación de error.

El funcionamiento de la unidad es el siguiente:

Los operandos de entrada X_1 y X_2 entran al procesador de datos, y al mismo tiempo pasan por los codificadores que generan los residuos correspondientes, que se llevan a los procesadores de residuos.

La salida del procesador de datos será el resultado de ejecutar la operación f_d sobre éstos (posiblemente incorrecto). Los procesadores de residuo tendrán como salida los que deben ser los residuos módulo m_1 y módulo m_2 del resultado de la operación.

Por ser tres unidades independientes, es muy pequeña la probabilidad de que falle mas de una a la vez. Así, podemos suponer que si hay error, lo habrá solamente en la salida de una de ellas, o bien en el procesador de datos, o bien en uno solo de los procesadores de residuo.

Estas tres salidas, entran en el generador de síndrome, del que se obtiene información suficiente para corregir el error. Como vimos en el capítulo 3, pueden presentarse cuatro casos con el síndrome:

$S = (0,0)$ ----- No hay error.

$S = (e,0)$ ----- Error en el residuo 1.

$S = (0,e)$ ----- Error en el residuo 2.

$S = (E_1,E_2)$ ----- Error en el dato.

Este síndrome, junto con el resultado de operar sobre los datos y los resultados de operar sobre los residuos, se introduce en el corrector, que se encarga de avisar si hubo o no error, calcular el valor de éste y restarselo al resultado errôneo para generar el resultado correcto. También

se encarga de generar los residuos correspondientes al resultado en el caso de que el error se haya producido en uno de los residuos.

Esta es la estructura genérica que debe adoptarse en una unidad de proceso protegida mediante un código birresiduo. La línea de puntos entre el procesador de datos y los procesadores de residuos corresponde a la información necesaria para determinar la operación f_r que hay que ejecutar sobre los residuos cuando sobre los datos se ejecuta f_d.

5.3.2. Duplicación interna.

Proponemos en este apartado una estrategia de protección para unidades de proceso, que es viable gracias a los avances vistos en el capítulo 2 de los niveles de integración de circuitos. La filosofía de ésta es ejecutar una detección y corrección de errores simultánea con el funcionamiento del sistema, pero utilizando para la detección una duplicación de unidades, y un código de residuos para la corrección, en lugar de utilizar, como la estrategia anterior un código para ambas funciones.

Los métodos de protección de unidades funcionales mediante duplicación se rechazaban en el pasado, puesto que el incremento del número de elementos lógicos empleados en un diseño se traducía directamente en un incremento proporcional en el costo de construcción del sistema. Actualmente, con la introducción de la integración de circuitos a gran escala, la economía es diferente: El incremento del costo en función del incremento del número de puertas se ha reducido mucho. Entonces, si el diseñador de un sistema quiere añadir circuitería extra en el circuito integrado para conseguir Tolerancia a Fallos, el incremento neto en el costo de fabricación del sistema será muy pequeño.

Sedmak y Liebergot (Sedmak 1980) propusieron la duplicación de unidades funcionales dentro de un circuito integrado para proteger su funcionamiento. De este modo es posible detectar los errores que se produzcan en dichos circuitos, y lanzar unas rutinas de recuperación.

La estructura de la unidad de proceso propuesta en este apartado es capaz de detectar y corregir de un modo simultáneo con el funcionamiento normal del sistema, cualquier error que se produzca en un solo circuito integrado de los que constituyen el sistema.

5.3.2.1. Unidad de proceso elemental.

Dividimos la unidad de proceso (como en la estrategia anterior) en rodajas de b bits de ancho, estando ahora cada rodaja duplicada en el interior de un circuito integrado. La estructura del circuito integrado que constituye una rodaja se muestra en la figura 5.13.

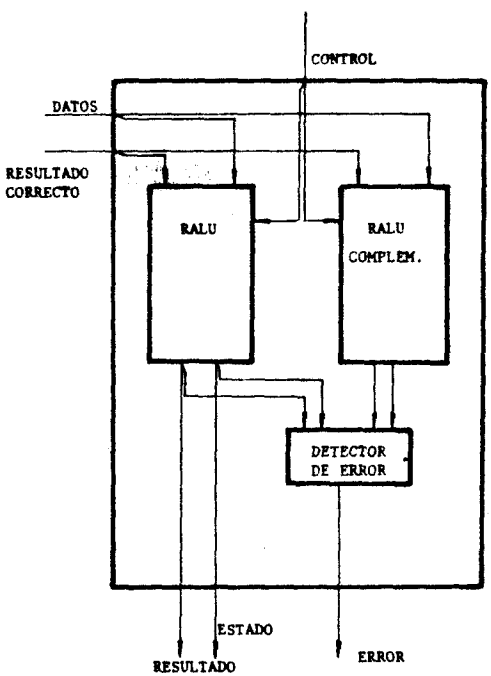


Figura 5.13

El área que llamamos RALU contiene el procesador aritmético y lógi-

co, además de un conjunto de registros de trabajo. Hay otro área de tamaño parecido que es una duplicación de la primera (debe construirse con lógica complementaria). Las entradas son comunes a ambos bloques, y las salidas de éstos se llevan a un comparador que genera una serie de señales de error codificadas, para lograr que dicho comparador sea autoverificado. Las entradas de alimentación y masa son redundantes, y la señal de reloj se lleva, ademas de a los bloques de RALU, a un comprobador que también genera una señal de error. De este modo, cualquier fallo que se produzca en el circuito integrado puede detectarse.

En la figura 5.14 se muestra un ejemplo de lógica complementaria. Las señales de entrada y de salida correspondientes tienen polaridades opuestas.

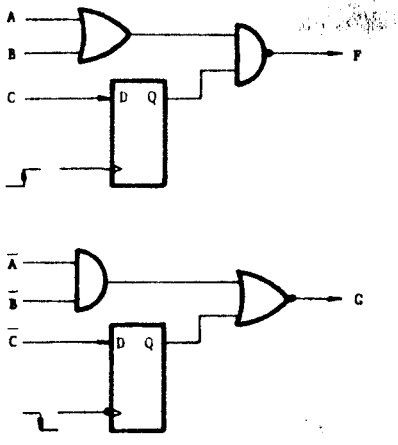


Figura 5.14

Si no utilizamos esta técnica de lógica complementaria, un fallo en la máscara durante el proceso de fabricación (que sería la misma para todas las RALU,s)podría provocar fallos idénticos en éstas, lo que daría lugar a

errores que podrían quedar indetectados.

El problema de que falle el detector de error del circuito integrado se resuelve codificando su salida por ejemplo en un código 1 de 2. Este código puede ser el siguiente:

00 ---- Error en el detector o en la RALU

01 ---- No error

10 ---- No error

11 ---- Error en el detector o en la RALU

La figura 5.15 muestra un comparador de dos bits que da su salida codificada de este modo, y que puede utilizarse para construir el detector.

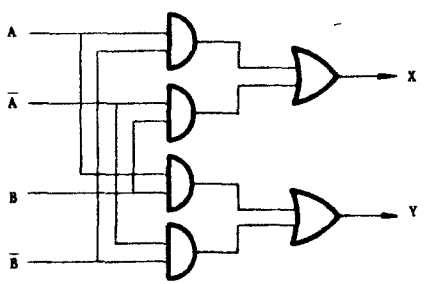


Figura 5.15

5.3.2.2. Constitución de la unidad de proceso.

Si organizamos la unidad de proceso a base de conectar rodajas del tipo propuesto, contaremos con la posibilidad de detectar todos los errores

que se produzcan. Además se puede localizar en cuál de las rodajas se ha producido éste. Para poder corregir el error falta conocer su magnitud para, restándola del dato, obtener el resultado correcto. Esto podemos lograrlo utilizando un código de resíduos, que a nivel físico supone añadir una rodaja más (algo distinta de las anteriores) y una unidad de corrección. La estructura resultante se representa en la figura 5.16.

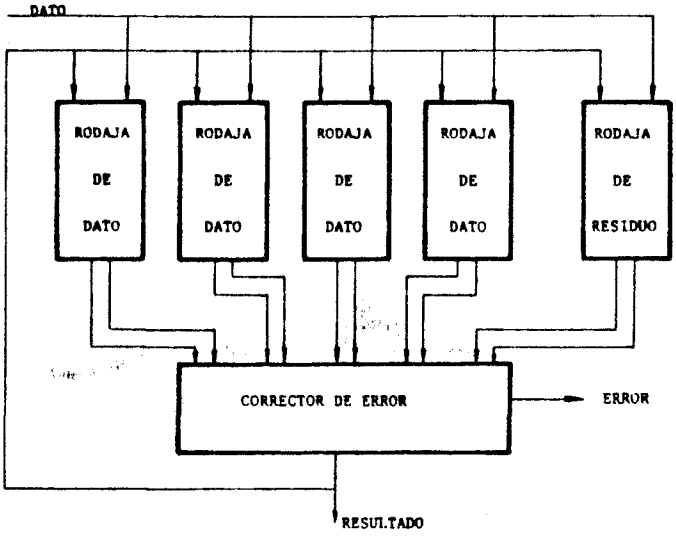


Figura 5.16

La rodaja de resíduo se encarga de generar el resíduo módulo 2^b-1 del dato de entrada y de procesar los resíduos de los operandos seleccionados, dando como salida el que debe ser el resíduo del resultado de procesar los datos en el resto de las rodajas. Para ello, debe contar internamente con la lógica adecuada para ejecutar la operación f_r correspondiente sobre los resíduos (aplicando los resultados obtenidos en el capítulo 4). Lógicamente, este circuito integrado también debe tener duplicada toda la unidad funcional, y debe contar con un detector de error que active las salidas

correspondientes.

El corrector de error también tiene las mismas características de duplicación que el resto de los circuitos integrados. Recibe como entradas las siguientes: El resultado de ejecutar la operación f_d sobre los datos, el resultado de ejecutar la operación f_r sobre los resíduos, y las salidas de error de todas las rodajas.

Esta unidad se encarga de generar el síndrome (calculando el resíduo módulo 2^b-1 del resultado de operar sobre los datos y restándolo del resultado de operar sobre los resíduos). Este síndrome representa la magnitud del error que se ha producido (si hubo alguno). De este modo, si se ha activado la señal de error de una sola rodaja y la magnitud del error no es cero, procederá a corregir el resultado (restándole la magnitud del error) para su posterior utilización o almacenamiento. Pueden presentarse tres casos:

1) El síndrome es nulo y no se activa la señal de error en ninguna de las rodajas. En este caso no se ha producido error, y por lo tanto no es necesaria la corrección.

2) El síndrome es distinto de cero y se activa la señal de error en una sola rodaja. En este caso, se ha producido error en la rodaja que activa la señal de error, y éste es corregible.

3) Independientemente de como sea el síndrome se activan señales de error en más de una rodaja. En este caso, detectamos el error, pero no es posible corregirlo porque éste es múltiple.

Vemos entonces que mediante el uso de esta estrategia podemos construir una unidad de proceso capaz de corregir todos los errores debidos a fallos en una sola rodaja, así como también detectar todos los errores debidos a fallos múltiples (en más de una rodaja). Asimismo cabe destacar que el código utilizado para la corrección es válido para datos de cualquier tamaño. Esto representa una ventaja sobre la estrategia anterior, en la que

aparte de tener que cumplir las bases de resíduo una serie de restricciones, si aumentábamos el rango de representación teníamos que variar dichas bases. Con esta segunda estrategia, para cualquier número de rodajas de b bits, basta utilizar un resíduo módulo 2^b-1.

5.3.3. Integración en un solo circuito.

Visto el avance previsto para los próximos años en la tecnología de integración de circuitos, cabe plantearse la construcción en un solo circuito integrado de una unidad de proceso Tolerante a Fallos. Vimos en el capítulo 2 que los fallos que se producen en un circuito integrado afectan a zonas determinadas del mismo, y ninguna zona es mas susceptible de fallo que otra. Por lo tanto, podemos utilizar para la protección de esta unidad la estrategia de dividirla en rodajas de b bits y utilizar un código birresíduo capaz de corregir todos los errores que se produzcan en una sola rodaja de dicho ancho.

Para proteger este circuito integrado se debe cuidar especialmente que la separación entre las diversas rodajas sea mayor que el area que se prevee que afectará el fallo. De este modo, nos aseguramos de que el fallo no afecte a mas de una rodaja. Teniendo en cuenta estas consideraciones e introduciendo redundancia en las líneas de conexión, se integrarán en un solo circuito tanto las rodajas de proceso de datos como las de proceso de resíduos y la unidad detectora y correctora de error.

Una segunda aproximación, para no perder tanta superficie en el circuito integrado sería no establecer las separaciones entre las diversas rodajas citadas anteriormente, sino integrar juntas todas las rodajas y utilizar un código que sea capaz de corregir un fallo que se produjera en dos rodajas contiguas. De este modo podríamos aumentar en gran manera el área permitida de fallo.

CAPITULO VI.

APLICACION.

En este capítulo abordamos el diseño de un computador con características de Tolerancia a Fallos, basándonos en los resultados obtenidos en los capítulos anteriores. Se realiza el diseño con circuitos integrados LSI y MSI de tecnología TTL LS, y se demuestra así la viabilidad de construir un computador con capacidad de corregir los errores que se produzcan en un solo circuito integrado.

La estructura general del computador diseñado se muestra en la figura 6.1.

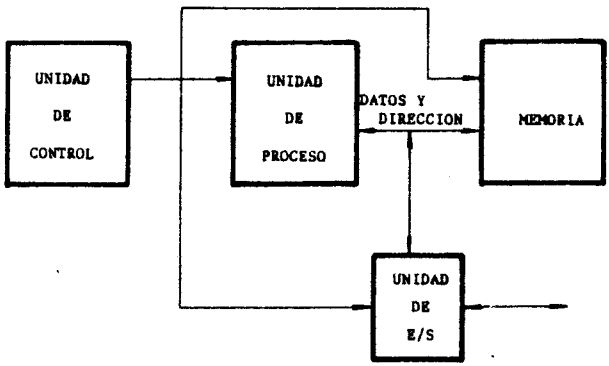


Figura 6.1

Este computador tiene la característica de que tanto los datos como las direcciones se encaminan por un bus único de 18 líneas (16 de datos y 2

de paridad para proteger la transmisión).

Tanto sobre la unidad de control como sobre la unidad de entrada salida, nos limitamos a hacer algunas consideraciones generales, centrándonos en el diseño de la memoria y de la unidad de proceso, cuya protección es el objetivo de este trabajo.

6.1. Unidad de control.

La unidad de control de este computador debe ser microprogramada, puesto que, de este modo, se pueden almacenar en memoria de control las rutinas de recuperación que necesite el sistema (su ejecución es mucho más rápida).

La figura 6.2 muestra la estructura general prevista para esta unidad de control.

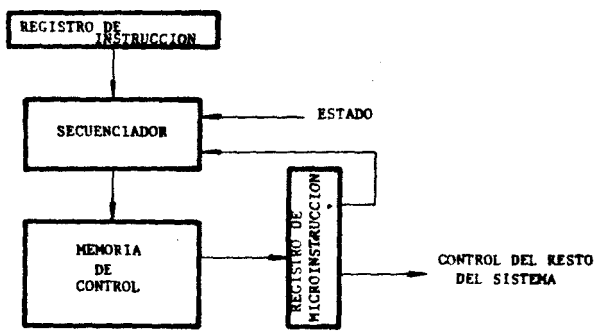


Figura 6.2

El único bloque de esta unidad que podemos proteger contra fallos mediante el uso de códigos detectores y correctores de error (objeto de nuestro trabajo) es la memoria de control. Esta se caracteriza por contar

con un número relativamente bajo de palabras de ancho muy grande (1 Kx64). Así pues, es conveniente organizarla en rodajas de b bits (en nuestro caso con pastillas de 1 Kx8, por lo cual utilizaremos un código corrector sobre $GF(2^b)$. El código idóneo para esta aplicación es el birredundante descrito en el capítulo 5, con el que podemos corregir cualquier error que se produzca en una sola rodaja.

El secuenciador de microinstrucciones, por no tener una estructura regular como la memoria, únicamente podemos protegerlo mediante replicación. Lo mejor es utilizar redundancia modular triple con decodificadores de mayoría (figura 6.3).

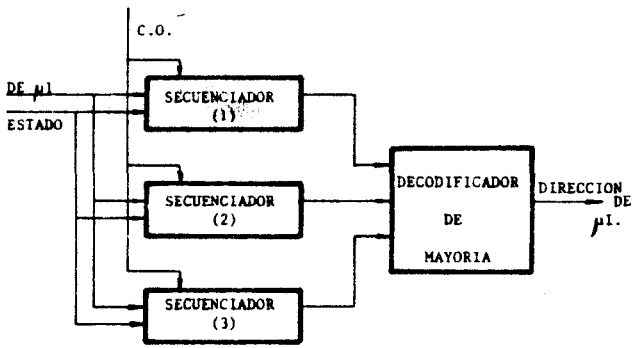


Figura 6.3

El mayor problema que podemos encontrarnos al diseñar este secuenciador es el de sincronizar las tres unidades a la hora de ejecutar la decodificación y decisión de cual es la dirección correcta. De todos modos, no es un problema insalvable.

Para proteger las propias señales de control de la máquina podemos utilizar un registro de microinstrucción, de tal modo que las señales que parten de la memoria de control se almacenan en el registro cuando han pa-

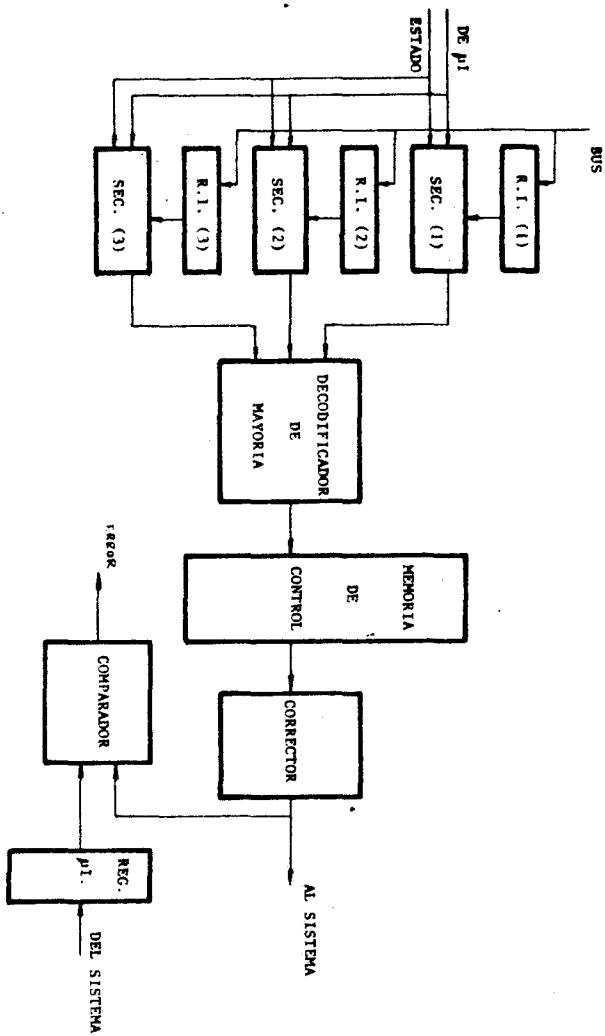

ESTADO
DE µI
BUS
SEC. (3)
R.I. (3)
SEC. (2)
R.I. (2)
SEC. (1)
R.I. (1)
DECODIFICADOR DE MAYORIA
MEMORIA DE CONTROL
CORRECTOR
COMPARADOR
ERROR
REG. µI.
AL SISTEMA
DEL SISTEMA

Figura 6.4

sado por todos los bloques donde tienen que actuar. Una comparación posterior del contenido del registro con la salida de la memoria podrá detectar si se ha perdido alguna señal de control, iniciando en este caso las acciones de recuperación oportunas.

Igualmente protegemos el código de operación mediante una triplicación del registro de instrucción. La unidad de control queda de un modo general como muestra la figura 6.4.

6.2. Memoria principal y unidad de proceso.

A la hora de diseñar la memoria y la unidad de proceso del computador cabe plantearse la posibilidad de utilizar el mismo código detector y corrector para proteger ambas unidades.

La utilización de un código de tipo paridad, que es muy útil en la memoria, es imposible en la unidad de proceso, debido a que no es cerrado respecto de las operaciones que se ejecutan en ésta (un código de tipo paridad solo es cerrado respecto de la operación "O" exclusivo).

En cambio, la utilización de un código aritmético para la memoria sí es viable, pero presenta el inconveniente de que la redundancia necesaria es mayor que con los códigos de tipo paridad. Entonces, la utilización en una memoria de un código aritmético llevaría a un aumento importante del número de elementos físicos que la componen. Por ello, en nuestro diseño hemos utilizado un código de tipo paridad en la memoria y un código aritmético en la unidad de proceso, evitando los transcodificadores de un modo muy simple: utilizando dos códigos sistemáticos, que permiten decodificar la palabra código mediante una simple truncación.

6.2.1. Memoria principal.

Como hemos concebido nuestro computador de modo que trabaje con datos de 16 bits, la memoria diseñada es capaz también de almacenar datos de 16 bits. Esta memoria cuenta con las siguientes características generales:

- Almacena 16 K-palabras de 22 bits (16 bits para datos y 6 bits redundantes).

- Cuenta con un protocolo de acceso muy simple basado en las señales DIR, ICM, R/W y DISP (dirección, inicio de ciclo de memoria, lectura/escritura y disponibilidad).

- La comunicación con la CPU se efectúa mediante un bus único para datos y direcciones de 16 bits, protegido mediante dos bits de paridad.

- Memoria dinámica con refresco asíncrono, es decir, que éste se efectúa automáticamente sin que intervenga la CPU.

- Capacidad para corregir cualquier error simple y para detectar cualquier error doble o múltiple de peso par. Para ello utiliza un código detector y corrector de error.

- La matriz de memoria está organizada en rodajas de un solo bit para aumentar la efectividad del código utilizado.

A continuación pasamos a describir el diseño de la unidad.

6.2.1.1 Descripción general.

En la figura 6.5 se muestra el diagrama general de la unidad de memoria. Cabe destacar los siguientes bloques:

- Matriz de memoria. (Figura 6.6) Está formada por una sola fila de circuitos integrados MM5290 (memoria de 16 Kx1) conectados con todas las líneas de dirección y de control en común. Solamente son independientes las líneas de dato (formando así 22 líneas de entrada y otras 22 de salida).

- Codificador detector de paridad. (figura 6.7) Basado en dos generadores de paridad de 9 entradas (74 LS 280), se encarga de generar dos bits de paridad para el envío de datos hacia la CPU, así como de detectar

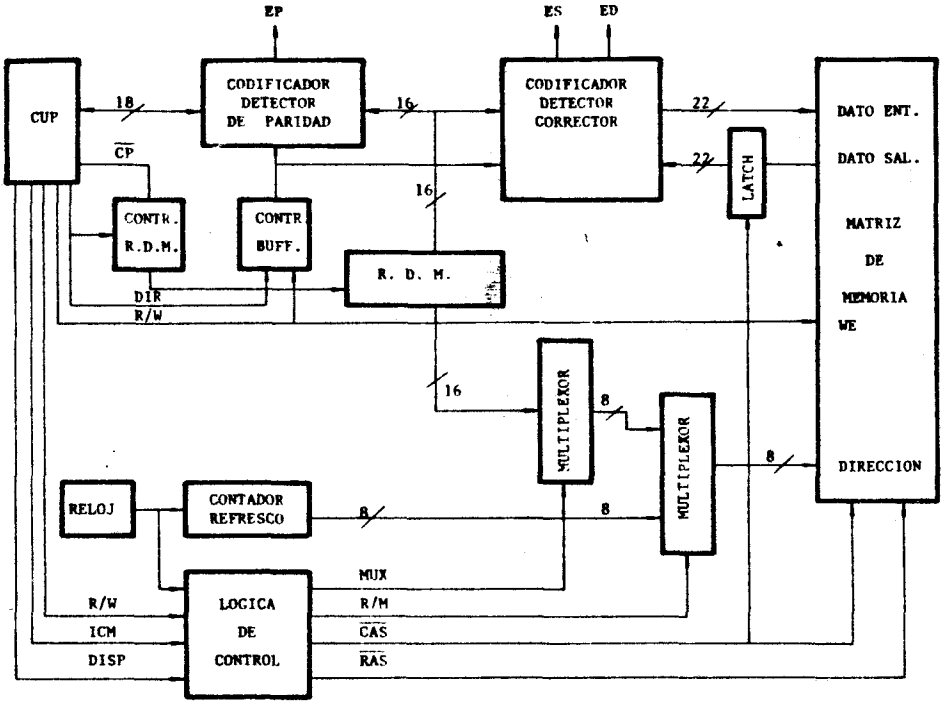

EP
ES
ED
CUP
18
CODIFICADOR DETECTOR DE PARIDAD
16
CODIFICADOR DETECTOR CORRECTOR
22
DATO ENT.
DATO SAL.
MATRIZ DE MEMORIA
WE
DIRECCION
CP
22
LATCH
16
CONTR. R.D.M.
CONTR. BUFF.
R. D. M.
DIR
R/W
16
MULTIPLEXOR
8
MULTIPLEXOR
8
RELOJ
CONTADOR REFRESCO
8
8
MUX
R/W
R/M
ICM
CAS
DISP
RAS
LOGICA DE CONTROL

Figura 6.5

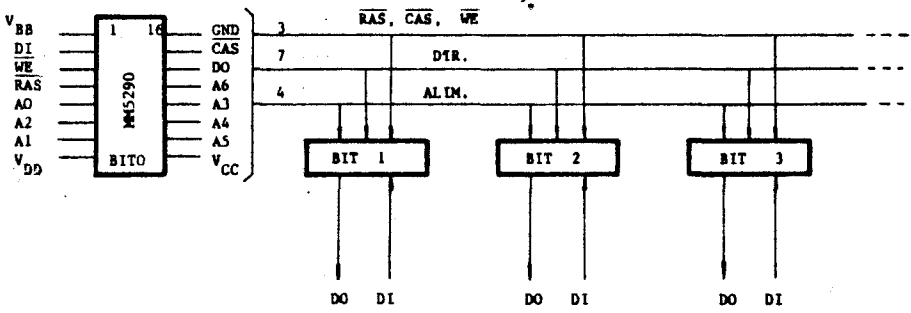


Figura 6.6

si se ha producido o no error en la transmisión de una dirección o un dato de la CPU a la memoria. Proporciona la salida de error en un código 1 de 2, verificando también su propio buen funcionamiento. Cabe destacar en su diseño el uso de "buffers" triestado para no duplicar el codificador en ambos sentidos de transmisión.

- Registro de direcciones de memoria (RDM). Está formado por dos circuitos integrados 74 LS 374 (registros de 8 bits) y se encarga de almacenar la dirección del dato que se quiere leer o escribir en un acceso.

- Codificador detector corrector (CDC). Se encarga de generar los seis bits redundantes que se almacenan junto con el dato en un acceso de escritura. Además se encarga este bloque de la generación del síndrome, de su decodificación y de la corrección del dato si se ha detectado un error simple en un acceso de lectura. Genera también las señales que indican a la CPU si se ha producido error simple o doble.

El código utilizado es de peso impar mínimo (22,16) de los discutidos en el capítulo 5. Su matriz de paridad H se muestra en la figura 6.8.

Está construida de acuerdo con el método descrito en el capítulo 5. De ella se obtiene la matriz generadora G, que se muestra en la figura 6.9.

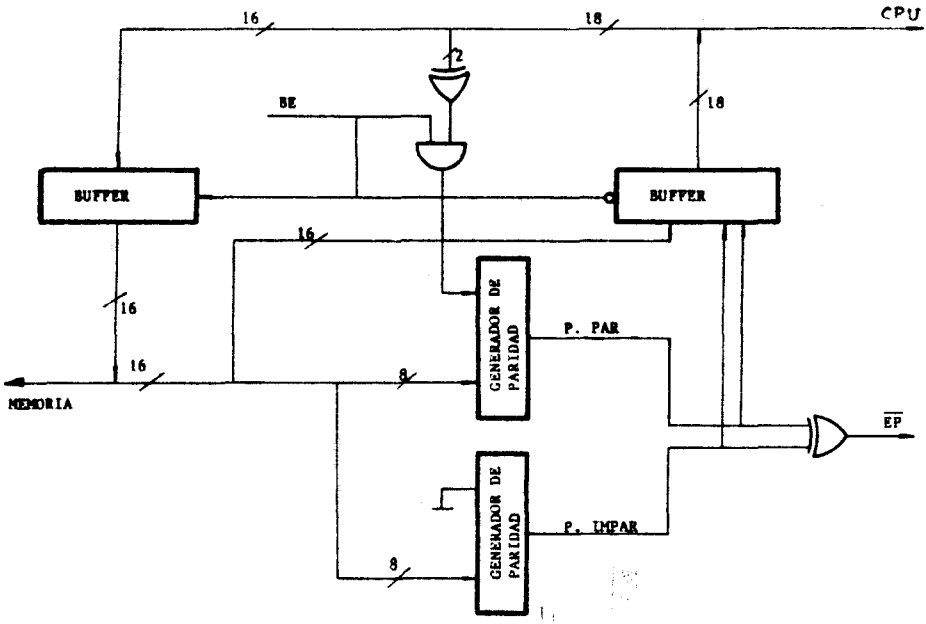


Figura 6.7

$$H = \begin{matrix} 1&1&1&1&1&1&1&0&1&0&0&0&0&0&0&1&0&0&0&0&0 \\ 1&1&1&1&0&0&0&1&0&1&1&1&0&0&0&0&1&0&0&0&0 \\ 1&0&0&0&1&1&1&1&0&0&0&0&1&1&1&0&0&0&1&0&0&0 \\ 0&1&0&0&1&0&0&0&1&1&1&0&1&1&0&1&0&0&0&1&0&0 \\ 0&0&1&0&0&1&0&0&1&1&0&1&1&0&1&1&0&0&0&0&1&0 \\ 0&0&0&1&0&0&1&1&0&0&1&1&0&1&1&1&0&0&0&0&0&1 \end{matrix}$$

Figura 6.8

De estas dos matrices obtenemos cómo debe diseñarse el CDC, obteniendo el mostrado en la figura 6.10.

$$G = \begin{matrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&1&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&1\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&1&1&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&1&0&1&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&0&1\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&1&0&0&1\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&1&1&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&1&1&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&1&0&1\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1
\end{matrix}$$

Figura 6.9

Se han utilizado buffers triestado para poder generar tanto el síndrome como los bits redundantes con el mismo codificador, formado por seis circuitos integrados 74 LS 280. La corrección se efectúa negando (mediante una puerta "O" exclusivo) el bit que se ha detectado y localizado como erróneo.

- Multiplexores de dirección. Se encargan de dividir la dirección de acceso en dos partes, dirección de fila y dirección de columna (esto es necesario por la utilización de memoria dinámica). Además entregan a la memoria la dirección de fila de refresco cuando se produce un ciclo de este tipo.

- Lógica de refresco. Está compuesta por un generador de reloj y un contador. Envía a la lógica de control una petición de ciclo de refresco (cuando es necesario) y genera la dirección de la fila en que hay que ejecutar dicho ciclo.

- Lógica de control. Genera todas las señales que necesita la matriz de memoria durante un ciclo de acceso, se encarga del protocolo de comunicación con la CPU, y además arbitra la coincidencia de peticiones de ciclo (memoria o refresco). Su diagrama se muestra en la figura 6.11.

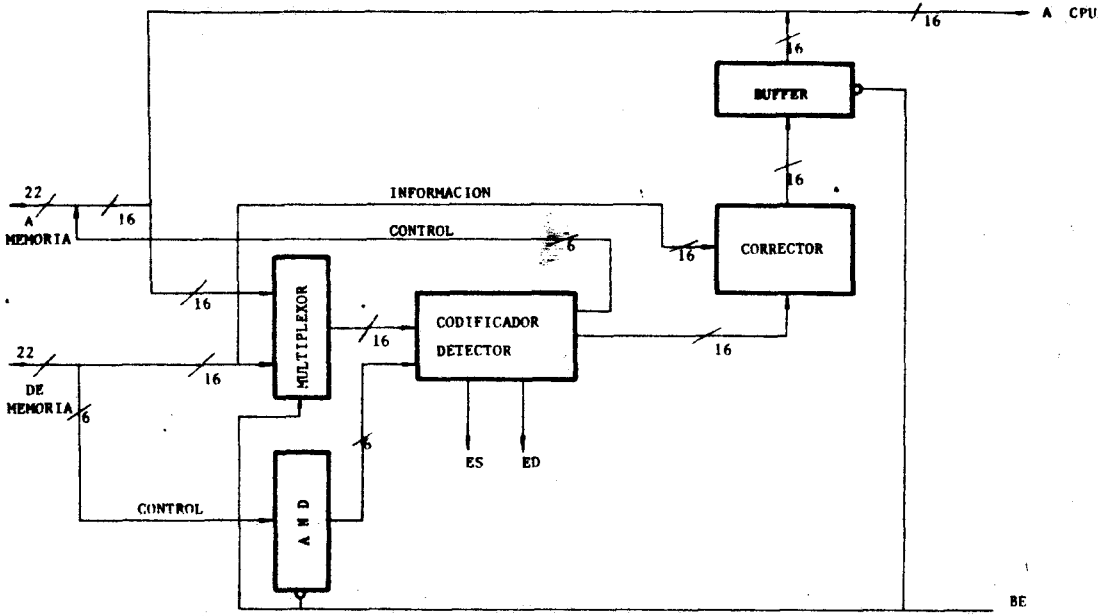

A CPU
16
16
BUFFER
16
INFORMACION
22
A
MEMORIA
16
CONTROL
6
16
CORRECTOR
16
MULTIPLEXOR
CODIFICADOR
DETECTOR
16
16
22
DE
MEMORIA
16
6
ES
ED
6
CONTROL
A N D
BE

Figura 6.10

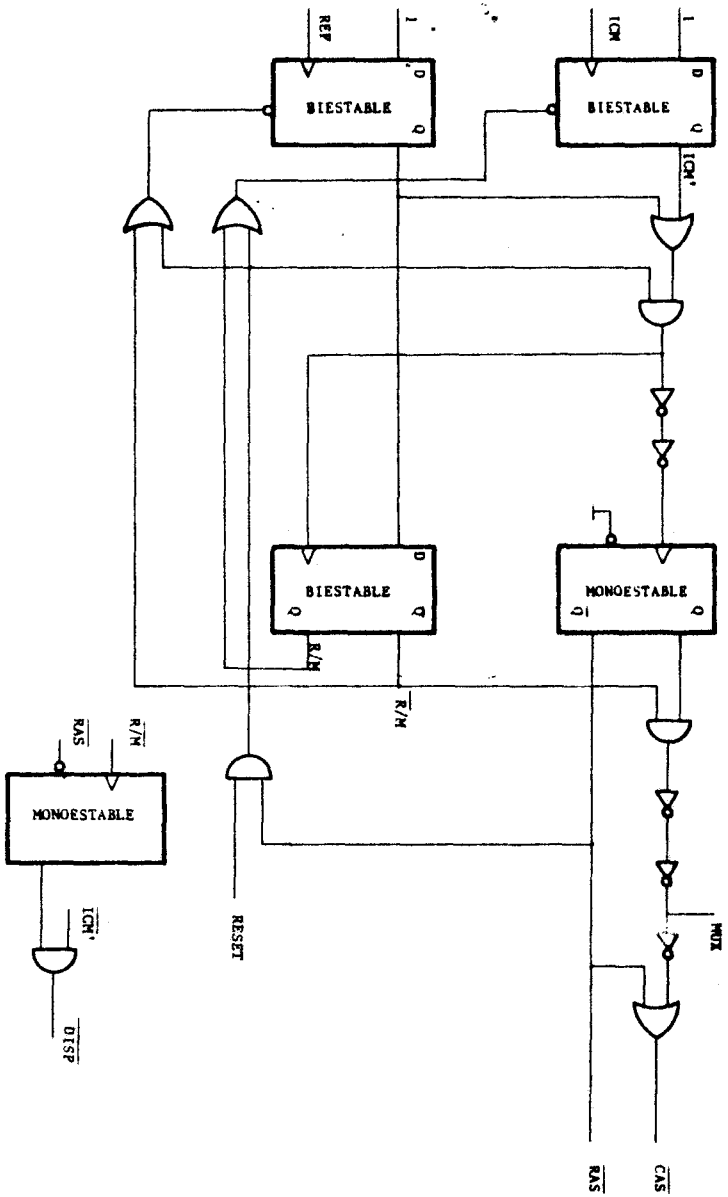

REF
I
ICM
I
BIESTABLE
BIESTABLE
ICM'
BIESTABLE
MONOESTABLE
R/M
R/M
RAS
R/M
MONOESTABLE
ICM'
DISP
RESET
MUX
RAS
CAS

Figura 6.11

Una vez descrita de un modo general la composición de la unidad de memoria, pasamos a describir a grandes rasgos su funcionamiento.

Durante la primera microistrucción de la rutina de acceso a memoria, la CPU pone en el bus de comunicación la dirección de acceso, al mismo tiempo que activa la señal DIR indicando a la memoria que lo que envía es una dirección. El codificador detector de paridad comprueba si se ha producido o no error en la transmisión, en cuyo caso activará la señal EP. La dirección queda a la entrada del RDM, donde se cargará con el flanco ascendente del reloj (enmascarado por la señal DIR).

En la siguiente microinstrucción, la CPU activa la señal ICM para solicitar el comienzo del ciclo, indica si quiere lectura o escritura con la señal R/W, y si es escritura, pone en el bus el dato a almacenar. Veamos los dos tipos de acceso.

Ciclo de lectura. Si no hay ninguna petición de ciclo de refresco pendiente o en ejecución, la señal ICM activa el ciclo de memoria en la lógica de control, y ésta baja la señal DISP, que no volverá a activarse hasta que termine el ciclo de acceso. Genera la señal $\overline{RAS}$, que introduce en la memoria la dirección de fila y a continuación cambia la señal MUX, poniendo a la entrada de dirección de la matriz de memoria la dirección de columna, que se carga cuando se activa la señal $\overline{CAS}$. Transcurrido el tiempo necesario para la lectura, el dato leído queda almacenado en un registro presente a la salida de la matriz de memoria, que lo mantiene a la entrada del CDC. En este bloque se genera el síndrome, se decodifica éste y se procede a la corrección del dato si se ha producido error (además de activar si es necesario las señales ES o ED). De aquí, los 16 bits de datos pasan al codificador de paridad, donde se generan los dos bits que se añaden para transmitir a la CPU el dato pedido. Por último, se activa la señal DISP para indicar a la CPU que se ha completado el acceso.

- Ciclo de escritura. Igual que en el de lectura, ICM activa el ciclo y obliga a que se desactive DISP. La dirección se carga del mismo modo en la memoria pero, como la señal R/W indica escritura, al mismo tiempo el

dato presente en el bus pasa por el codificador detector de paridad, que indica si hubo o no error en la transmisión. Los 16 bits de datos pasan por el CDC, generándose los seis bits redundantes, que se almacenan en la memoria junto con los 16 de datos. Cuando se completa el ciclo de escritura vuelve a activarse la señal DISP.

Por el hecho de haber utilizado en esta implantación física circuitos integrados de memoria dinámica, para que no se pierda la información almacenada, hay que ejecutar un ciclo de refresco en cada fila cada cierto intervalo de tiempo (en nuestro caso hay que refrescar todas las filas cada 2 mS). Como son 128 las filas a refrescar, basta con lanzar un ciclo de refresco cada 15 μS. De esto se encarga el reloj de refresco, cuyo flanco ascendente incrementa el contador de fila de refresco y lanza una petición de ciclo. Este será un ciclo de solo $\overline{RAS}$ (no se activa la señal $\overline{CAS}$), teniendo seleccionada como entrada de dirección de la memoria la salida del contador de refresco.

6.2.2. Unidad de proceso.

La unidad de proceso diseñada y construida para nuestro computador cuenta con las siguientes características:

- Está construida con tecnología TTL LS.

- Trabaja sobre operandos de 25 bits (16 bits de dato y 9 bits redundantes).

- Está dividida en 6 rodajas verticales, 4 para procesar los datos (cada una de ellas de un ancho de 4 bits) y otras dos (una de 4 bits y otra de 5) para procesar los bits redundantes que le dan su capacidad de corrección de errores.

- Cuenta con 16 registros de trabajo organizados como una memoria de acceso aleatorio con dos puertas de acceso.

- Elige los operandos A y B entre las siguientes fuentes: a) Registros de trabajo. b) Dato externo. c) Cero.

- Sobre estos operandos puede ejecutar las operaciones siguientes:

* Preset
* Clear
* Suma A+B
* Diferencia A-B
* Diferencia B-A
* "Y" Lógico A * B
* "O" Lógico A v B
* "O" Exclusivo A ⊕ B
* Todo tipo de desplazamientos y rotaciones

- Mediante la utilización de un código aritmético, es capaz de corregir cualquier error que se produzca en una sola rodaja, ya sea de datos o de bits redundantes, así como en el bit de acarreo, en el de cero o en el de signo. También es capaz de detectar la mayor parte de los errores múltiples.

Describimos primero el código utilizado para a continuación describir el diseño de la unidad.

6.2.2.1. Código utilizado.

Hemos utilizado para proteger la unidad de proceso un código birresiduo de los discutidos en el capítulo 5, que es capaz de corregir todos los errores que se produzcan en una sola rodaja de cuatro bits. Hemos elegido como bases de residuo las siguientes: $m_1 = 15$ y $m_2 = 31$. Este código tiene capacidad para trabajar con 20 bits de dato, que nosotros hemos restringido a 16 para trabajar con un tamaño de palabra standard.

El hecho de haber elegido las bases de residuo de la forma 2^c-1 ($m_1 = 2^4-1$ y $m_2 = 2^5-1$) hace que, tanto la codificación como el cálculo del

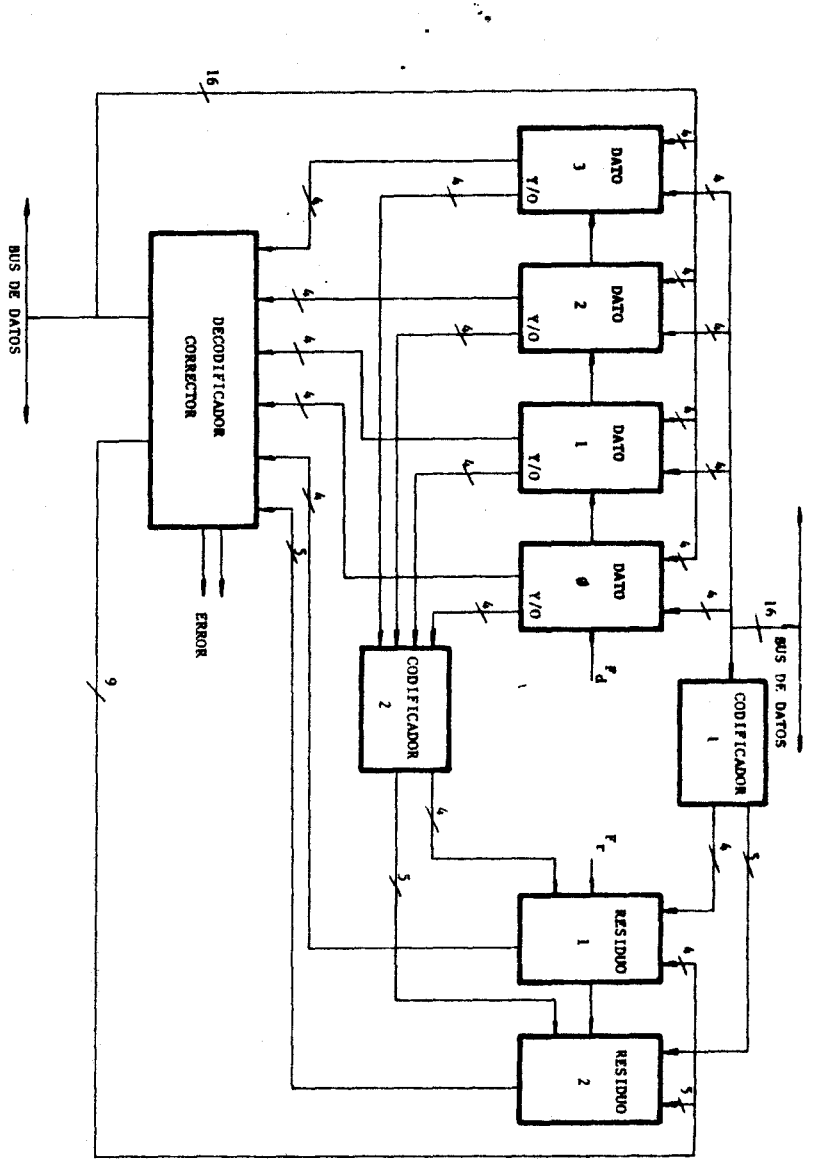

BUS DE DATOS
DECODIFICADOR CORRECTOR
ERROR
DATO 3 T/O
DATO 2 T/O
DATO 1 T/O
DATO 0 T/O
CODIFICADOR 2
CODIFICADOR 1
RESIDUO 1
RESIDUO 2
BUS DE DATOS
16
9
4
5

Figura 6.12

síndrome resulten muy simples (código de bajo costo). Además, por ser la primera rodaja de residuo del mismo ancho que las rodajas de datos, la parte del síndrome correspondiente a ésta nos da exactamente la magnitud del error, si éste se ha producido en una sola rodaja. La posición y el signo del error lo determinamos decodificando el síndrome completo. De este modo, también la corrección puede hacerse fácilmente, puesto que bastará sumar al dato erróneo una palabra formada por la primera parte del síndrome (invertida y colocada en la posición de la rodaja en que se ha localizado el error) y el resto puesto a unos o a ceros según indique el signo de dicho error. Así obtenemos el resultado correcto.

En la tabla de la página 188 pueden verse los posibles errores, junto con sus síndromes correspondientes.

6.2.2.2 Descripción general.

En esta unidad de proceso, cuyo diagrama general se muestra en la figura 6.12, cabe destacar los siguientes bloques:

1.- Codificador 1.

A la hora de introducir un dato en la unidad de proceso, hay que proceder a su codificación, es decir, a formar la palabra código con la que se va a trabajar. Esta consiste en enviarlo sin modificar a la parte de dato, y calcular su residuo módulo 15 y módulo 31 en el codificador para enviarlo a las rodajas de residuo. Por haber elegido las bases de la forma 2^c-1, es decir, código de bajo costo, podemos construir el codificador a base de árboles de sumadores. Para generar el residuo módulo 15, utilizamos tres sumadores módulo 15 (4 bits con realimentación de acarreo), y para generar el residuo módulo 31, tres sumadores módulo 31 (5 bits). El codificador es el mostrado en la figura 6.13.

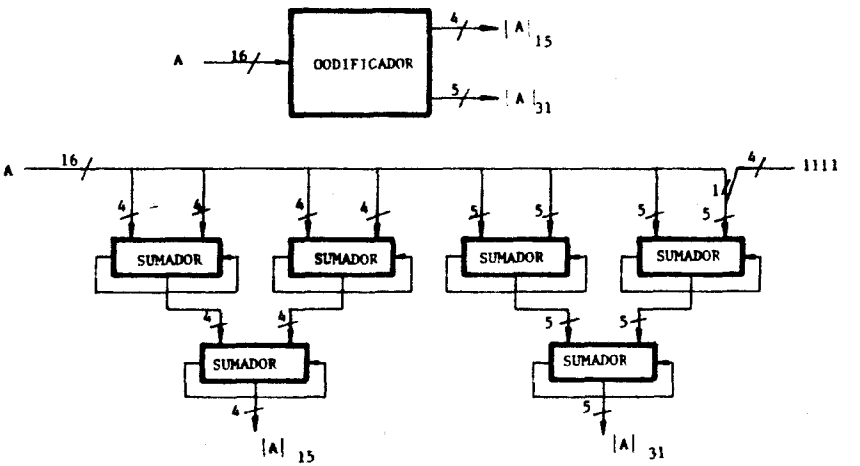


Figura 6.13

2.- Rodajas de datos.

Cuenta con cuatro rodajas iguales de cuatro bits para procesar los datos. Cada una de ellas responde al diagrama de bloques de la figura 6.14 y tiene las siguientes características:

* Registros. Cuenta con 16 registros de trabajo de 4 bits organizados como una memoria de acceso aleatorio con dos puertas de acceso. Por ambas puertas puede direccionarse el mismo registro. Necesita cuatro señales por cada puerta para direccionamiento, una señal de reloj y otra para inhibición de escritura. Estos registros están integrados en un solo circuito.

* Lógica de selección de operandos. Se encarga de poner a la entrada del operador los operandos A y B, seleccionando cada uno de ellos entre las siguientes fuentes: a) Registro de trabajo direccionado por la puerta A (R_A). b) Registro de trabajo direccionado por la puerta B (R_B). c) Dato ex-

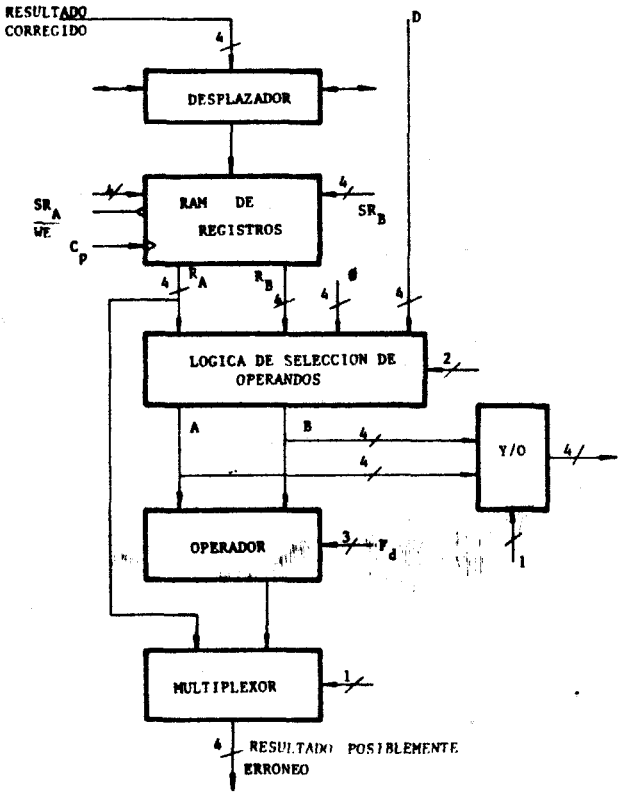


Figura 6.14

terno. d) Cero. Esta lógica, construida a base de multiplexores, está gobernada por cuatro señales de control.

* Operador. Está integrado en un solo circuito, y es capaz de ejecutar suma, diferencia, "Y", "O" y "O" exclusivo sobre los operandos A y B, además de PRESET y CLEAR. Se selecciona la operación a ejecutar mediante tres líneas de control.

* Multiplexor de salida. Selecciona como salida de la rodaja o bien el resultado de la operación, o bien el registro direccionado por la puerta A.

* Desplazador. A este bloque le llega el resultado de la operación ya corregido con el fin de no transmitir errores de una rodaja a otra. Está construido a base de multiplexores e introduce en el registro seleccionado el resultado sin desplazar, desplazado a la derecha o desplazado a la izquierda. Se controla mediante dos señales.

* Bloque "Y"/"O". Duplica el cálculo de una de estas dos operaciones lógicas con el fin de complementar el uso del código corrector de error en la protección de las operaciones lógicas. Está controlado por una de las líneas de control del operador.

3.- Rodajas de residuo.

La unidad de proceso cuenta con dos de estas rodajas, una de 4 bits para procesar el residuo módulo 15, y otra de 5 bits para procesar el residuo módulo 31. La estructura de ambas se muestra en la figura 6.15.

En estas dos rodajas de residuo se han implantado físicamente los resultados obtenidos en el capítulo 4 para mantener el código cerrado respecto de todas las operaciones que es capaz de ejecutar la unidad de proceso. Esto se logra haciendo corresponder a la operación f_d ejecutada sobre los datos, una f_r a ejecutar en estas rodajas. Para ello, además del código f_d, se introduce en éstas información procedente de la parte de datos sobre el acarreo de salida, además del residuo de la duplicación del "Y" o del "O" lógico.

La estructura de las rodajas de residuo es similar a la de las rodajas de datos, siendo iguales (salvo en ancho) la RAM de registros, la lógica de selección de operandos y el multiplexor de salida. A continuación describimos el resto de los bloques.

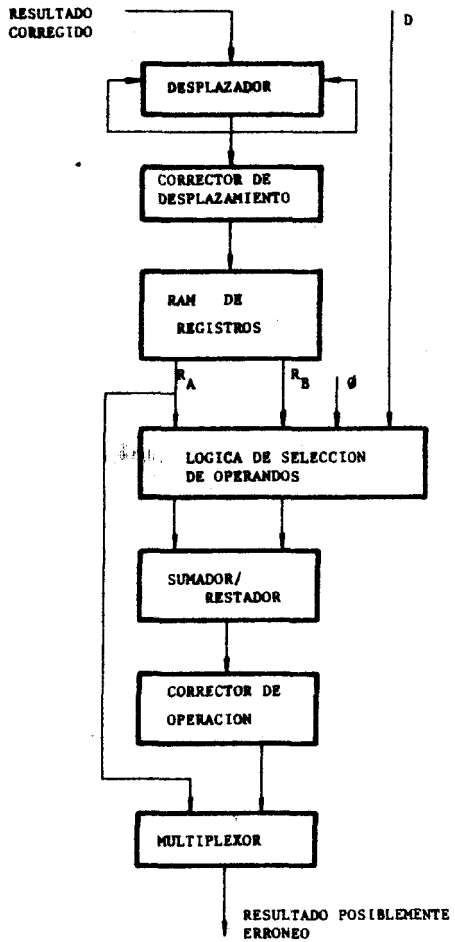

RESULTADO
CORREGIDO
D
DESPLAZADOR
CORRECTOR DE
DESPLAZAMIENTO
RAM DE
REGISTROS
R_A
R_B
Ø
LOGICA DE SELECCION
DE OPERANDOS
SUMADOR/
RESTADOR
CORRECTOR DE
OPERACION
MULTIPLEXOR
RESULTADO POSIBLEMENTE
ERRONEO

Figura 6.15

* Desplazador. Construido a base de multiplexores, está conectado de modo que, cuando se ordena desplazamiento, siempre ejecuta rotación (a izquierda o derecha). Su salida se lleva al corrector de desplazamiento para tener en cuenta tanto el bit que se introduce para desplazar el dato, como el que se pierde por el otro extremo.

* Corrector de desplazamiento. Obtuvimos en el capítulo 4 que la operación f_r correspondiente a un desplazamiento a la derecha de los datos es una rotación a la derecha del residuo seguida de una corrección, consistente en sumar la cantidad $|X_n|2^{n-1}|_{m_i} - X_0 2^{a-1}|_{m_i}$. Asimismo, la operación f_r correspondiente a un desplazamiento a la izquierda es una rotación a la izquierda del residuo seguida de la corrección consistente en sumar la cantidad $|X_{-1} - X_{n-1}|2^n|_{m_i}|_{m_i}$. En estas dos expresiones es:

X_n---- Bit que se introduce por la izquierda

X_0---- Bit que va a perderse por la derecha

a ---- $m_i = 2^a - 1$

X_{-1}---- Bit que se introduce por la derecha

X_{n-1}--- Bit que se pierde por la izquierda

Así pues, como a la entrada de este módulo llega el residuo con la rotación ya hecha, basta que sumemos la corrección correspondiente. Vamos a determinar cual es la corrección necesaria en cada uno de los casos y a mostrar el diseño de este bloque para las dos rodajas.

Residuo módulo 15.

Para $m_i = 15$, es $a = 4$. Por tanto,

$$|X_{16}|2^{15}|_{15} - X_0 2^3|_{15} = |8X_{16} - 8X_0|_{15}$$

$$|X_{-1} - X_{15}|2^{16}|_{15}|_{15} = |X_{-1} - X_{15}|_{15}$$

Poniendo en binario este resultado:

Desplazamiento a la derecha:

X_{16}	X_0	Corrección
0	0	0000 o 1111
0	1	0111
1	0	1000
1	1	0000 o 1111

De este modo, la cantidad a sumar para la corrección de un desplazamiento a la derecha es la cadena $X_{16}X_0X_0X_0$.

Desplazamiento a la izquierda:

X_{-1}	X_{15}	Corrección
0	0	0000 o 1111
0	1	1110
1	0	0001
1	1	0000 o 1111

La cantidad a sumar para ejecutar la corrección en un desplazamiento a la izquierda es la cadena $X_{15}X_{15}X_{15}X_{-1}$.

Así pues, el corrector de desplazamiento en la rodaja de residuo módulo 15 es el de la figura 6.16.

Residuo módulo 31.

Para $m_i = 31$ es $a = 5$. Por tanto,

$$|X_{16}|2^{15}|_{31} - X_0\,2^4|_{31} = |X_{16} - 16X_0|_{31}$$
$$|X_{-1} - X_{15}|2^{16}|_{31}|_{31} = |X_{-1} - 2X_{15}|_{31}$$

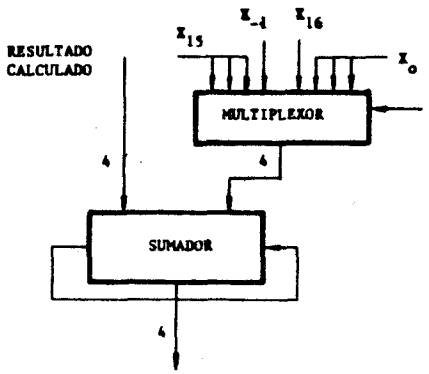


Figura 6.16

Pasándolo a binario queda:

Desplazamiento a la derecha:

X_{16}	X_0	Corrección
0	0	00000 o 11111
0	1	01111
1	0	00001
1	1	10000

Entonces, la corrección necesaria para el desplazamiento a la derecha es la cadena:

$$(X_0 \oplus X_{16}), \overline{X}_{16}, \overline{X}_{16}, \overline{X}_{16}, (\overline{X_0 X_{16}})$$

Desplazamiento a la izquierda:

X_{-1}	X_{15}	Corrección
0	0	00000 o 11111
0	1	11101
1	0	00001
1	1	11110

De modo que la corrección a sumar para el desplazamiento a la izquierda es la cadena: .

$$X_{15},X_{15},X_{15},(X_{-1}X_{15}),(X_{-1}\oplus X_{15}).$$

Por tanto, el corrector de desplazamiento en la rodaja de residuo módulo 31 es el que se muestra en la figura 6.17.

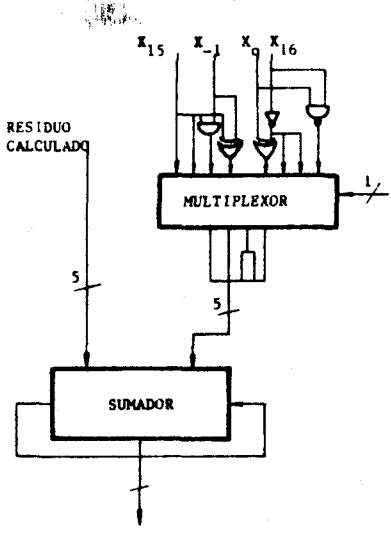


Figura 6.17

* Operador. En este caso, el operador está construido con un sumador con realimentación de acarreo (módulo 15 en un caso y módulo 31 en el otro), además de la lógica de inversión de operando para ejecutar las restas y la lógica para PRESET y CLEAR. No se prevee la posibilidad de ejecutar las operaciones lógicas directamente, puesto que en estas rodajas de residuo dichas operaciones se ejecutan mediante una suma y una corrección como veremos. La salida de este bloque se lleva a la entrada del corrector de operación que se encarga de ajustar el resultado.

* Corrector de operación. Se encarga de corregir el resultado de la operación ejecutada sobre los residuos (para mantener el código cerrado) teniendo en cuenta el acarreo de salida producido al operar sobre las partes de dato (si la operación es suma o diferencia) y el residuo de la duplicación del "Y" o del "O" lógico (si se trata de una operación lógica).

Respecto de las operaciones suma, diferencia, "Y", "O" y "O" exclusivo se obtuvieron en el capítulo 4 los resultados que se muestran en la tabla 6.1.

Operación sobre datos	Operación sobre residuos	
	Operación	Corrección
Suma	Suma	$-C_n\|2^n\|_{m_i}$
Diferencia	Diferencia	$(1-C_n)\|2^n\|_{m_i}$
"Y" lógico	Suma	$\|V(X_1 \; X_2)\|_{m_i}$
"O" lógico	Suma	$\|V(X_1 \; X_2)\|_{m_i}$
"O" exclusivo	Suma	$\|2V(X_1 \; X_2)\|_{m_i}$

Tabla 6.1

Vamos a particularizar estos resultados para las bases de residuo consideradas $m_1 = 15$ y $m_2 = 31$, pasando a continuación a describir el diseño del corrector de operación.

Debido a que el operador de esta unidad se ha construido a base de un sumador con realimentación de acarreo, hay que tener en cuenta también en el corrector de operación el acarreo de entrada para operaciones de suma y diferencia. Las correcciones quedan como sigue:

Residuo módulo 15.

Suma. La corrección que hay que ejecutar es $-C_n|2^n|_{m_1}$, a la que hay que añadir el acarreo de entrada C_0. Así, la corrección para la suma es la cantidad $|-C_{16}|2^{16}|_{15} + C_0|_{15}$, o lo que es lo mismo, $|-C_{16} + C_0|_{15}$. Pasándolo a binario, tenemos:

C_{16}	C_0	Corrección
0	0	0000 o 1111
0	1	0001
1	0	1110
1	1	0000 o 1111

Diferencia. La corrección a ejecutar es $(1 - C_n)|2^n|_{m_1}$. Hay que restarle el complemento del acarreo de entrada $(1 - C_0)$. Así, la corrección queda:

$$|(1 - C_n)|\, 2^n|_{m_1} - (1 - C_0)|_{15} =$$

$$= ||(1 - C_{16})|2^{16}|_{15} - (1 - C_0)|_{15} =$$

$$= |-C_{16} + C_0|_{15}.$$

Al pasar a binario resulta una tabla idéntica a la anterior. Así pues, para ambas, suma y diferencia, la corrección consiste en sumar la cadena: $C_{16}C_{16}C_{16}C_0$.

Incluyendo también la correccion para operaciones lógicas, el corrector de operación en la rodaja de residuo módulo 15 es el que muestra la figura 6.18.

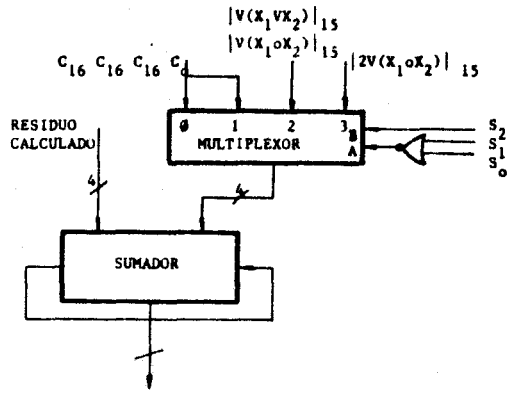


Figura 6.18

Residuo módulo 31.

Suma. La corrección es:

$$|-C_n|2^n|_{m_1} + C_0|_{m_1} = |-C_{16}|2^{16}|_{31} + C_0|_{31} =$$

$$= |-2C_{16} + C_0|_{31}$$

Pasando a binario queda:

C_{16}	C_0	Corrección
0	0	00000 o 11111
0	1	00001
1	0	11101
1	1	11110

Diferencia. La corrección es:

$$|(1 - C_n)|2^n|_{m_i} - (1 - C_0)|_{m_i} =$$

$$= |(1 - C_{16})|2^{16}|_{31} - (1 - C_0)|_{31} =$$

$$= |2(1 - C_{16}) - (1 - C_0)|_{31} = |1 - 2C_{16} + C_0|_{31}$$

Pasando a binario obtenemos:

C_{16}	C_0	Corrección
0	0	00001
0	1	00010
1	0	11110
1	1	00000 o 11111

Mezclando ambas tablas queda:

Suma con C_0=0
Diferencia con C_0=1 Corrección: $C_{16}C_{16}C_{16}C_{16}\overline{C}_{16}$

Suma con C_0=0
Diferencia con C_0=1 Corrección: $C_{16}C_{16}C_{16}C_0C_{16}$

De este modo, introduciendo también la corrección de operaciones lógicas, el corrector de operación de la rodaja de residuo módulo 31 es el mostrado en la figura 6.19

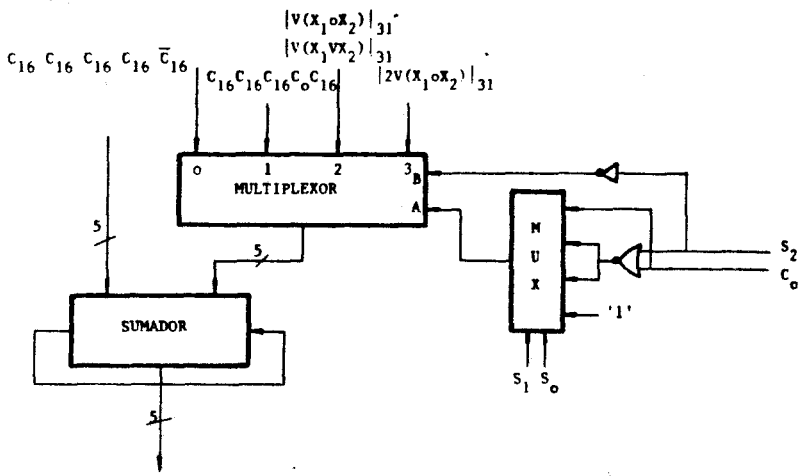


Figura 6.19

4.- Codificador 2.

Tiene exactamente la misma estructura que el codificador 1, y se encarga de calcular el residuo módulo 15 y módulo 31 de la duplicación de las operaciones lógicas para entregarlo a las rodajas de residuo.

5.- Decodificador/Corrector.

Esta unidad, partiendo del resultado de la operación sobre los datos (posiblemente incorrecto) y del resultado de operar sobre los residuos, comprueba si se ha producido o no error, y corrige dicho resultado para su posterior almacenamiento y utilización. Su diagrama de bloques se muestra en la figura 6.20.

Está constituido por los siguientes bloques:

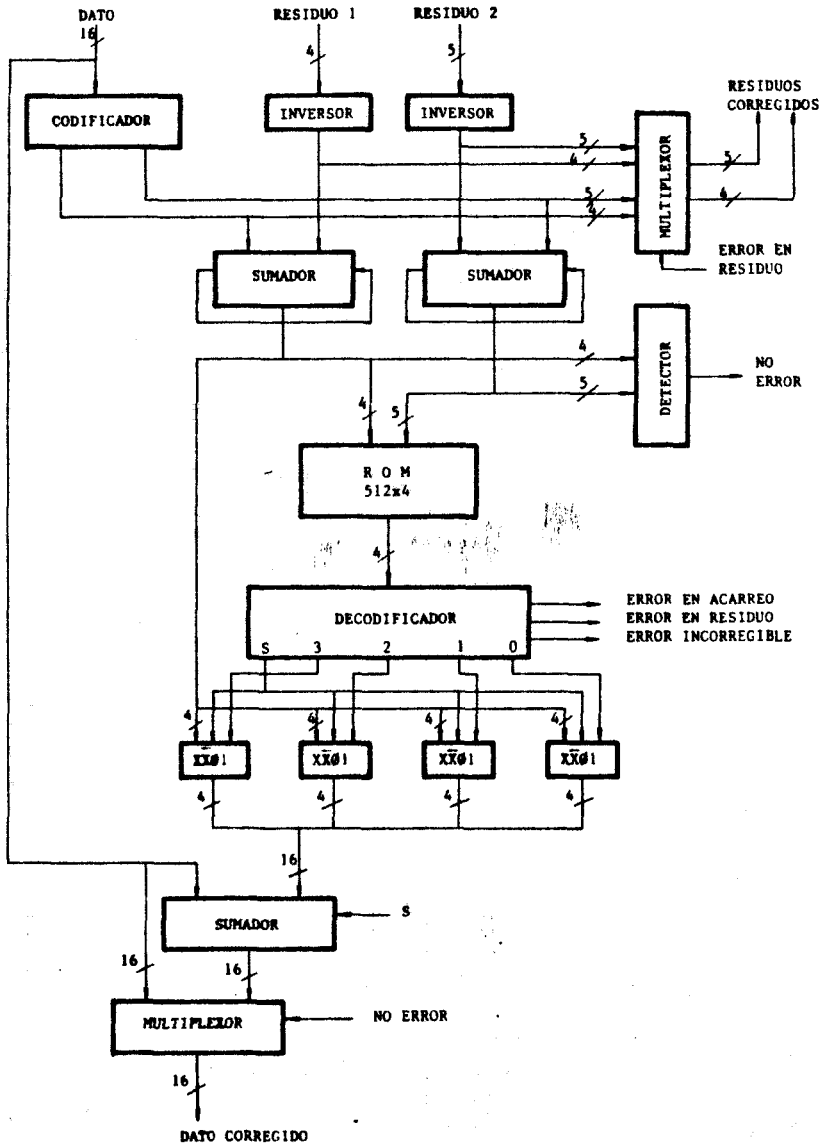

DATO
16
RESIDUO 1
4
RESIDUO 2
5
CODIFICADOR
INVERSOR
INVERSOR
RESIDUOS
CORREGIDOS
MULTIPLEXOR
ERROR EN
RESIDUO
SUMADOR
SUMADOR
DETECTOR
NO
ERROR
R O M
512x4
DECODIFICADOR
S 3 2 1 0
ERROR EN ACARREO
ERROR EN RESIDUO
ERROR INCORREGIBLE
SUMADOR
S
MULTIPLEXOR
NO ERROR
DATO CORREGIDO

Figura 6.20

* Codificador. Recibe el resultado de operar sobre los datos y calcula sus residuos módulo 15 y módulo 31 para calcular el síndrome restando de estos los resultados de operar sobre los residuos.

* Decodificador de síndrome. El síndrome va por una parte a un detector que se encarga de avisar si se ha producido o no error, y por otra a una memoria ROM de 512 palabras de 4 bits. En la ROM se ha codificado el significado de cada uno de los posibles síndromes, y el decodificador conectado a su salida se encarga de generar las señales que indican de que tipo es el error que se ha producido, en cual de las rodajas y con que signo. La magnitud del error la dan directamente los cuatro bits del síndrome correspondientes al residuo módulo 15.

* Localizador de error. Formado por 4 circuitos integrados (rodajas de 4 bits) en cuya salida ponen o bien su entrada, el negado de ésta, cuatro unos o cuatro ceros, se encarga de generar la palabra de 16 bits que hay que sumar al dato erróneo para obtener el dato correcto.

* Corrector. Consta únicamente de un sumador de 16 bits.

Si no se produce error, mediante un multiplexor se pone sin modificar en la salida de este bloque el dato que entró. Si se detecta error en los residuos, tampoco se modifica el dato, y los residuos que salen del bloque para almacenarlos son los calculados a partir del dato.

6.-Funcionamiento.

El funcionamiento de esta unidad de proceso puede resumirse como sigue:

Bajo control de microprograma, se seleccionan los operandos, que aparecen a la entrada del operador (tanto en las rodajas de dato como en las de residuo), y la operación f_d. Transcurrido el tiempo oportuno, el operador presenta a su salida el resultado de la operación, posiblemente erróneo (por fallo en el operador o en los operandos). El dato resultante

se lleva al codificador donde se generan los residuos módulo 15 y módulo 31, y se calcula el síndrome teniendo en cuenta el resultado obtenido en las rodajas de residuo. Si este síndrome es cero, se activa la señal de NO ERROR, y del corrector salen el dato y los residuos originales para su almacenamiento y utilización. Si el síndrome es distinto de cero, se alarga el ciclo de reloj para dar tiempo a la corrección. Se decodifica el síndrome y se obtiene el tipo de error ocurrido. Pueden darse los siguientes casos:

- Error en el dáto. Se determina en cual de las rodajas se ha producido, se forma la palabra de corrección y se suma ésta al resultado de la operación para obtener el resultado correcto.

- Error en el residuo. El dato resultante se saca sin modificación y como residuos se ponen en la salida los obtenidos a partir de dicho dato resultante.

- Error solo en el acarreo de salida. Se saca sin modificar el dato resultante, se complementa el acarreo y se corrigen los residuos, puesto que se ha ejecutado mal la corrección de operación en las rodajas de residuo.

- Error incorregible. En este caso se detecta fallo en más de una rodaja. Esto ha podido producirse por dos razones: 1) Los dos operandos son erróneos en la misma rodaja. 2) Hay fallo en más de una rodaja.

Cuando detectamos error incorregible se inhibe la carga de registros (para mantener en éstos los datos anteriores) y se genera una interrupción. La microrrutina correspondiente debe sacar uno de los operandos y almacenarlo corregido. A continuación se ejecuta de nuevo la operación que falló, y si falla de nuevo, estamos seguros de que el error se ha producido en dos rodajas (caso 2). El error es realmente incorregible. En caso contrario, se ha logrado corregir el error.

A la hora de proteger las operaciones lógicas, se han duplicado dos

de ellas ("Y" y "0") para incluir en la corrección los operandos, puesto que si se hubiese duplicado solamente una, seríamos capaces unicamente de corregir los fallos que se produjeran en el operador.

V(E)	Sindr.	V(E)	Sindr.	V(E)	Sindr.	V(E)	Sindr.
0001	E 1E	FFFE	1 01	0010	E 0F	FFEF	1 10
0002	D 1D	FFFD	2 02	0020	D 1E	FFDF	D 0F
0003	C 1C	FFFC	3 03	0030	C 0E	FFCF	3 11
0004	B 1B	FFFB	4 04	0040	B 1D	FFBF	4 02
0005	A 1A	FFFA	5 05	0050	A 0D	FFAF	5 12
0006	9 19	FFF9	6 06	0060	9 1C	FF9F	6 03
0007	8 18	FFF8	7 07	0070	8 0C	FF8F	7 13
0008	7 17	FFF7	8 08	0080	7 1D	FF7F	8 04
0009	6 16	FFF6	9 09	0090	6 0B	FF6F	9 14
000A	5 15	FFF5	A 0A	00A0	5 1A	FF5F	A 05
000B	4 14	FFF4	B 0B	00B0	4 0A	FF4F	B 15
000C	3 13	FFF3	C 0C	00C0	4 19	FF3F	C 06
000D	2 12	FFF2	D 0D	00D0	2 09	FF2F	D 16
000E	1 11	FFF1	E 0E	00E0	1 18	FF1F	E 07

V(E)	Sindr.	V(E)	Sindr.	V(E)	Sindr.	V(E)	Sindr.
0100	E 17	FEFF	1 08	1000	E 1B	EFFF	1 04
0200	D 0F	FDFF	2 10	2000	D 17	DFFF	2 08
0300	C 07	FCFF	3 18	3000	C 13	CFFF	3 0C
0400	B 1E	FBFF	4 01	4000	B 0F	BFFF	4 10
0500	A 16	FAFF	5 09	5000	A 0B	AFFF	5 14
0600	9 0E	F9FF	6 11	6000	9 07	9FFF	6 18
0700	8 06	F8FF	7 19	7000	8 03	8FFF	7 1C
0800	7 1D	F7FF	8 02	8000	7 1E	7FFF	8 01
0900	6 15	F6FF	9 0A	9000	6 1A	6FFF	9 05
0A00	5 0D	F5FF	A 12	A000	5 16	5FFF	A 09
0B00	4 05	F4FF	B 1A	B000	4 12	4FFF	B 00
0C00	3 1C	F3FF	C 03	C000	3 0E	3FFF	C 11
0D00	2 14	F2FF	D 0B	D000	2 0A	2FFF	D 15
0E00	1 0C	F1FF	E 13	E000	1 06	1FFF	E 19

CAPITULO VII.

CONCLUSIONES.

Las conclusiones mas importantes que se desprenden de este trabajo son las siguientes:

- Visto el aumento de los campos de aplicación de los computadores, se plantea la necesidad de usar técnicas de Tolerancia a Fallos en aplicaciones en que tanto la fiabilidad como la seguridad de funcionamiento son críticas.

- Se efectúa un análisis y tipificación de los modos de fallo en los circuitos integrados, que junto con la capacidad de integración de la tecnología condicionan la forma de particionar un computador, así como las estrategia óptimas de Tolerancia a Fallos.

- Se hace una recopilación, sistematización y puesta al dia de los códigos correctores de error mas interesantes para su aplicación en el diseño de computadores. Esto nos sirve además de base para plantear las estrategias de detección y corrección de error.

- Se plantean y demuestran 8 teoremas inéditos que nos permiten resolver el problema de mantener cerrado un código de residuos respecto de todas las operaciones elementales, sin tener que introducir ninguna restricción en las bases de residuo ni en el tipo de aritmética utilizada (complemento a 1 o complemento a 2). Esto permite optimizar el diseño.

- Se plantean varias estrategias de detección y corrección de error en funcionamiento simultáneo con el propio sistema, eligiendo las estrategias óptimas de acuerdo con su mayor velocidad, menor redundancia y simplicidad de implantación física. Esto se logra aplicanco los resultados ante-

riores a la problemática concreta del diseño de computadores.

-Se plantea el diseño, construcción y prueba de un prototipo construido de acuerdo con una de las estrategias planteadas. Debe destacarse la sencillez de resolución de los problemas de implantación, comprobándose en la práctica la validez de las estrategias presentadas, así como su posibilidad de industrialización.

- Cabe destacar que se ha obtenido un beneficio lateral de la utilización de estrategias de detección y corrección de error, puesto que gracias a ésto, la puesta a punto ha podido realizarse de un modo muy rápido y simple.

- El empleo de esta estrategia de corrección de error aumenta el costo de la unidad aproximadamente un 100 %, puesto que el número de puertas necesario es aproximadamente el doble del que se ne necesitaría en una unidad sin redundancia.

- En este trabajo se abre como campo de investigación futura la continuación en la línea de estudio de unidades aritméticas Tolerantes a Fallos viendo en concreto la integración de toda la unidad en un solo circuito como se cita en el capítulo 5.

Otra línea de trabajo prevista es completar un minicomputador Tolerante a fallos diseñando y construyendo las dos unidades que faltan, es decir, la unidad de control y la unidad de entrada-salida.

CAPITULO VIII.

REFERENCIAS.

(Ashjaee 1976)

M.J. Ashjaee; S.M. Reddy.
On totally self checking checkers for separable codes.
Int. Simp. on Fault Tolerant Computing. 1976. pp. 151-156.

(Avizienis 1971)

A. Avizienis.
The STAR (Self Testing And Repairing) computer: An investigation of the theory and practice of Fault Tolerant computer design.
IEEE Trans. on Computers. C-20, 11. Noviembre, 1971. pp. 1312-1321.

(Avizienis 1971)

A. Avizienis.
Arithmetic error codes: Cost and effectiveness studies for application in digital system design.
IEEE Trans. on Computers. C-20, 11. Noviembre 1971. pp. 1322-1331.

(Avizienis 1976)

A. Avizienis.
Fault Tolerant Systems.
IEEE Trans. on Computers. C-25, 12. Diciembre 1976. pp 1304-1312.

(Avizienis 1977)

A. Avizienis.
Fault Tolerant Computing: Progress, problems and prospects.
IFIP 1977. pp. 405-420.

(Avizienis 1978)

A. Avizienis.

Fault Tolerance: The survival attribute of digital systems.

IEEE Proceedings. V-66, 10. Octubre 1978. pp. 1109-1125.

(Avizienis 1978)

A. Avizienis.

Fault Tolerance in computer systems.

INFOTECH 1978. pp. 39-67.

(Avizienis 1978)

A. Avizienis.

Computer systems reliability: An overview.

INFOTECH. 1978. pp. 24-35.

(Ayache 1979)

J.M. Ayache; M. Diaz.

A reliability model for error correcting memory systems.

IEEE Trans. on Reliability. R-28, 4. Octubre 1979. pp. 310-315.

(Bennets 1979)

R.G. Bennets.

Fault Tolerance and digital systems.

Microprocess. and Microsystems (G.B.). V-3, 8. Octubre 1979. pp. 365-373.

(Bennets 1979)

R.G. Bennets.

A review of Fault Tolerance and its application to digital systems containing VLSI components.

Proc. of Microtest. 1979 (England). pp. 1-18.

(Berlekamp 1968)

E.R. Berlekamp.

Algebraic coding theory.

McGraw-Hill. 1968.

(Berlekamp 1980)

E.R. Berlekamp.

The technology of error correcting codes.

IEEE Trans. on Computers. V-66, 5. Mayo 1980. pp. 564-593.

(Bhandarkar 1979)

D.P. Bhandarkar.

The impact of semiconductor technology on computer systems.

IEEE Computer. Septiembre 1979. pp. 92-98.

(Bhargava 1978)

V.K. Bhargava.

Some codes of Rahman and Blake for computer applications.

IEEE Trans. on Computers. C-27, 8. Agosto 1978. pp. 765-767.

(Black 1977)

C.J. Black et al.

Development of a spaceborne memory whith a single error and erasure correction scheme.

Int. Symp. on Fault Tolerant Computing. 1977. pp. 50-55.

(Bose 1977)

A.K. Bose; S.A. Szygenda.

Design of a diagnosable and Fault Tolerant Input-output controller.

National Computer Conference. 1977. pp. 795-800.

(Bossen 1970)
D.C. Bossen.
B-adyacent error correction.
IBM J. Res. Develop. V-14, Julio 1970. pp. 402-408.

(Bossen 1976)
D.C. Bossen; L.C. Chang.
Package detectability of error correcting codes.
Int. Symp. on Fault Tolerant Computing. 1976. pp 206.

(Bossen 1978)
D.C. Bossen et al.
Measurement and generation for error correcting codes for package failures.
IEEE Trans. on Computers. C-27, 3. Marzo 1978. pp. 201.

(Carter 1970)
W.C. Carter; D.C. Jessep; A. Wadia.
Error free decoding for faillure tolerant memories.
IEEE Proc. ICGC. Junio, 1970. pp. 229-239

(Carter 1971)
W.C. Carter; K.A. Duke; D.C. Jessep.
A simple self testing decoder checking circuit.
IEEE Trans. on Computers. C-20, 11. Noviembre 1971. pp. 1413-1414

(Carter 1975)
W.C. Carter; C.E. McCarthy.
Implementation of an experimental Fault Tolerant memory system.
Int. Symp. on Fault Tolerant Computing. 1975. pp. 113-118

(Carter 1976)

W.C. Carter.

Implementation of an experimental Fault Tolerant memory system.

IEEE Trans. on Computers. C-25, 6. Junio 1976. pp. 557-568.

(Carter 1977)

W.C. Carter et al.

Cost effectiveness of self checking computer design.

Int. Symp on Fault Tolerant Computing. 1977. pp. 117-123.

(Cenker 1979)

R.P. Cenker et al.

A Fault Tolerant 64K dynamic Random Acces Memory.

IEEE Trans. on Electron Devices. ED-26, 6 Junio 1979. pp. 853-860.

(Cliff 1980)

R.A. Cliff.

Acceptable testing of VLSI components which contains error correctors.

IEEE Trans. on Computers. C-29, 2. Febrero 1980. pp. 125-134.

(Cook 1973)

R.W. Cook et al.

Design of self checking microprogram controls.

IEEE Trans. on Computers. C-22, 3. Marzo 1973. pp. 255-262.

(Cox 1978)

G.W. Cox; B.D. Carroll.

Reliability modeling and analysis of Fault Tolerant memories.

IEEE Trans. on Reliability. R-27, 1. Abril 1978. pp. 49-54.

(Chinal 1975)

J.P. Chinal.

An error analysis of loop free iterative adders for digit complement code.

Int. Symp. on Fault Tolerant Computing. 1975. pp. 143-148.

(Davida 1970)

G.I. Davida; J.P. Robinson.

Detection-correction decoding for system reliability.

IEEE Trans. on Reliability. R-19, 4. Noviembre, 1970. pp. 188-190

(Devries 1979)

R.C. Devries.

Comments on "A readily implemented single error correcting unit distance counting code.

IEEE Trans. on Computers. C-28, 3. Marzo 1979. pp. 253-261.

(Diaz 1975)

M. Diaz; J. Moreira de Souza.

Design of self checking microprogrammed controls.

Int. Symp. on Fault Tolerant Computing. 1975. pp. 137-142.

(Easterbrook 1977)

J.T. Easterbrook; R.G. Bennets.

Failure mechanisms in logic circuits and their related fault effects.

New Dev. in Automatic Testing (IEEE London). 1977. pp. 44-47

(ED. 1980)

New era for LSI: Specialized circuits.

Electronic Design. 15 Febrero 1980. pp. 41-52.

(Foley 1979)

E. Foley.

The effects of the microelectronics revolution on system and board test.

IEEE Computer. Octubre 1979. pp. 32-38.

(Fujiwara 1980)

E. Fujiwara.

Error control for byte package organized memory systems.

Trans. of IECE of Japan. E-63, 2. Febrero 1980. pp. 98-103.

(Gabet 1979)

J.M. Gabet.

VLSI: The impacts grows.

Datamation. Junio 1979. pp. 109-112.

(Gaddes 1970)

T.G. Gaddes.

An error detecting binary adder: A hardware shared implementation.

IEEE Trans. on Computers. C-19, 1. Enero, 1970. pp.34-38

(Galiay 1980)

J. Galiay et al.

Phisical versus logical fault models MOS LSI circuits : Impact on their testability.

IEEE Trans. on Computers. C-29, 6. Junio 1980. pp. 527-531.

(Garcia 1968)

O.N. Garcia; T.R.N. Rao.

On the methods checking logical opperations.

2nd. annual Conference on Information Sciences and systems.

Marzo, 1968. pp.89-95.

(Garner 1966)

H.L. Garner.

Error codes for arithmetic operations.

IEEE Trans. on Electron Computers. EC-15. 1966. pp. 763-770

(Goldberg 1974)

J. Goldberg; K.N. Levitt; J.H. Wensley.

An organization for highly survivable memory.

IEEE Trans. on Computers. C-23, 7. Julio 1974. pp. 693-705.

(Goldberg 1980)

J. Goldberg.

SIFT: A probable Fault Tolerant Computer for aircraft flight control.

Information Processing 80. IFIP 1980. pp. 151-156.

(Hart 1980)

A. Hart et al.

Reliability influences from electrical overstress on LSI devices.

Ann. Proc. on Reliability Phisics. 1980. pp. 190-196.

(Hartwell 1978)

W.T. Hartwell; C.W. Hoffner; W.N. Toy.

A Fault Tolerant memory for duplex system.

IEEE Trans. on Reliability. R-27, 2. Junio 1978. pp. 134-138.

(Heller 1977)

W.R. Heller et al.

Prediction of wiring space requirements for LSI.

Design Automation Conference (N. Orleans) 1977. pp. 32-42.

(Holton 1980)

W. Holton; G. Brown.

Potential barriers to Very Large Scale Integration.

Proc. Conferene on Computing in the 1980's. pp.213-218.

(Hong 1972)

S.H. Hong.

A general class of maximal codes for computer applications.

IEEE Trans. on Computers. C-21, Diciembre 1972. pp. 1322-1331.

(Hsiao 1970)

M.Y. Hsiao.

A class of optimal minimum odd-weight-column SEC/DED codes.

IBM J. Res. Develop. V-14. Julio 1970. pp. 395-401

(Hsiao 1975)

M.Y. Hsiao; D.C. Bossen.

Orthogonal latin square configuration for LSI memory yield and reliability enhancement.

IEEE Trans. on Computers. C-24, 5. Mayo 1975. pp. 513-516.

(Hwang 1976)

K. Hwang.

Designing reliable microprogram control whith partitioned hibrid redundancy

Int. Symp. on Fault Tolerant Computing. 1976. pp. 45-51.

(Kameyama 1980)

M. Kameyama; T. Higuchi.

Design of dependent Failure Tolerant microcomputer system using triple modular redundancy.

IEEE Trans. on Computers. C-29, 2 Febrero 1980. pp. 202-206.

(Keyes 1975)

R.W. Keyes.

Phisical limits in digital electronics.

IEEE Proceedings. V-63. Mayo 1975.

(Keyes 1979)

R.W. Keyes.

The evolution of digital electronics towards VLSI.

IEEE Trans. on Electron Devices. ED-26, 4. Abril 1979. pp. 271-279.

(Kime 1980)

C.R. Kime.

Fault Tolerant Computing: An introduction and a perspective.

IEEE Trans. on Computers. C-29, 5. Mayo 1980. pp. 457-460.

(Koczela 1971)

L.J. Koczela.

A three faillure tolerant computer system.

IEEE trans. on Computers. C-20, 11. Noviembre 1971. pp.1389-1393

(Levine 1976)

L. Levine; W. Meyers.

Semiconductor memory reliability with error detecting and correcting codes.

IEEE Computer. V-9. Octubre 1976. pp. 43-50.

(Lin 1970)

S. Lin.

An introduction to error correcting codes.

Prentice Hall 1970.

(Marouf 1978)

M.A. Marouf; A.D. Friedman.

Design of self checking checkers for Berger codes.

Int. Symp. on Fault Tolerant Computing. 1978. pp. 179-184.

(Massey 1964)

J.L. Massey.

Survey of residue coding for arithmetic errors.

International Comput. Center Bull. 1964.

(Matsuzawa 1977)

K. Matsuzawa; Y. Tohma.

A way of multiple error correction for computer main memory.

Trans. on Inst. Electron and Commun. Japan. E-60, 10 Octubre 1977.

pp. 603-604.

(May 1979)

T.C. May.

Soft errors in VLSI. Present and future.

Electronic components conference. 1979. pp. 247-256.

(McCarthy 1975) C.E. McCarthy; W.C. Carter; J.B. White.

A memory system which can tolerate multiple storage array faults.

Southeastern Symp. on System Theory. 1975. pp.172-178.

(McPartland 1980)

R.J. McPartland et al.

Alpha particle induced soft errors and 64K dynamic RAM design interaction.

Ann. Proc. on Reliability Phisics. 1980. pp. 261-267.

(McWilliams 1977)

F.J. McWilliams; N.J.A. Sloane.

The theory of error correcting codes. Part I.

North-Holland. 1977.

(McWilliams 1977)

F.J. McWilliams; N.J.A. Sloane.
The theory of error correcting codes. Part II.
North-Holland. 1977.

(Neches 1979)

P.M. Neches.
VLSI architecture, design and fabrication.
IEEE Computer. Mayo 1979. pp. 76-78.

(Neumann 1973)

P.G. Neumann; T.R.N. Rao.
Error correction in byte organized arithmetic processors.
Int. Symp. on Fault Tolerant Computing. Junio 1973. pp. 53-58.

(Neumann 1975)

P.G. Neumann; T.R.N. Rao.
Error correcting codes for byte organized arithmetic processors.
IEEE Trans. on Computers. C-24, 3. Marzo 1975. pp. 226-232.

(Peeples 1980)

J.W. Peeples; T.J. Every.
Parametric influences on system soft error rates.
Ann. Proc. on Reliability Phisics. 1980. pp. 255-260

(Peterson 1961)

W.W. Peterson.
Error correcting codes.
MIT Press. 1961.

(Peterson 1978)

W.W. Peterson; E.J. Weldon.
Error correcting codes.
MIT Press. 1978.

(Pickel 1978)

J.C. Pickel; J.T. Blandford.

Cosmic ray induced errors in MOS memory cells.

IEEE Trans. on Nuclear science. NS-25. Diciembre 1978.

(Pradhan 1974)

D.K. Pradhan.

Syntesis of Fault Tolerant arithmetic and logic processors by using nonbinary codes.

Int. Symp. on Fault Tolerant Computing. Junio 1974. pp. 4.22-4.28.

(Pradhan 1974)

D.K. Pradhan.

Fault Tolerant carry save adders.

IEEE Trans. on Computers. C-23, 11. Noviembre 1974. pp. 1320-1322.

(Pradhan 1980)

D.K. Pradhan.

A New class of error correcting-detecting codes for Fault Tolerant Computer applications.

IEEE Trans. on Computers. C-29, 6. Junio 1980. pp. 471-481.

(Ramamoorthy 1971)

C.V. Ramamoorthy.

Fault Tolerant Computing: An introduction and an overview.

IEEE Trans. on Computers, C-20, 11. Noviembre 1971.

(Rao 1968)

T.R.N. Rao.

Use of error correcting codes on memory words for improved reliability.

IEEE Trans. on Reliability. R-17, 2. Junio 1968. pp.91-96.

(Rao 1968)

T.R.N. Rao.

Error checking logic for arithmetic type operations of a processor.

IEEE Trans. on Computers. C-17, 9. Septiembre 1968. pp. 845-849.

(Rao 1972)

T.R.N. Rao.

Error correction in adders using systematic subcodes.

IEEE Trans. on Computers. C-21, 3. Marzo 1972. pp. 254-259.

(Rao 1974)

T.R.N. Rao.

Error coding for arithmetic processors.

Academic Press. 1974.

(Rao 1977)

T.R.N. Rao; H.J. Reinheimer.

Fault Tolerant modularized arithmetic logic units National Computer Conference. 1977. pp. 703-710.

(Rennels 1978)

D.A. Rennels.

Architectures for Fault Tolerant spacecraft computers.

IEEE Proceedings. V-66, 10. Octubre 1978.

(Rennels 1978)

D.A. Rennels et al.

A study of standard building blocks for the design of Fault Tolerant distributed computer systems.

Int. Symp. on Fault Tolerant Computing. 1978. pp. 144-149.

(Rickers 1980)

H.C. Rickers; P.F. Manno.

Microprocessor and LSI microcircuit reliability prediction model.

IEEE Trans. on Reliability. R-29, 3. Agosto 1980. pp. 196-202.

(Sahni 1970)

R.J. Sahni.

Reliability of integrated circuits.

IEEE Proc. Int. Computer Group Conf. Wasington D.C. Junio, 1970. pp.213-219

(Sedmak 1978)

R.M. Sedmak; H.L. Liebergot.

Fault Tolerance of a general purpose computer implemented by very large scale integration.

Int. Symp. on Fault Tolerant Computing. 1978. pp. 137-143.

(Sedmak 1980)

R.M. Sedmak; H.L. Liebergot.

Fault Tolerance of a general purpose computer implemented by very large scale integration.

IEEE Trans. on Computers. C-29, 6. Junio 1980. pp. 492-500.

(Siewiorek 1978)

D. Siewiorek; S. Elkind.

The effect of semiconductor memory chip failure modes on system reliability and performance.

Int. Symp. on Fault Tolerant Computing. 1978. pp. 150-155.

(Smith 1977)

J.E. Smith; G. Metze.

The design of totally self checking combinational circuits.

Int. Symp. on Fault Tolerant Computing. 1977. pp. 130-134.

(Srinivasan 1971)

C.V. Srinivasan.

Codes for error correction in high speed memory systems. Part I: Corrections of cell defects in integrated memories.

IEEE Trans. on Computers. C-20, 8. Agosto 1971. pp.882-888.

(Srinivasan 1971)

C.V. Srinivasan.

Codes for error correction in high speed memory systems. Part II: Correction of temporary and catastrophic errors.

IEEE Trans. on Computers. C-20, 12. Diciembre, 1971. pp. 1514-1520.

(Stiffler 1976)

J.J. Stiffler.

Architectural design for near 100% fault coverage.

Int. Symp. on Fault Tolerant Computing. 1976. pp. 134-137.

(Stiffler 1978)

J.J. Stiffler.

Coding for Random Acces Memories

IEEE Trans. on Computers. C-27, 6. Junio 1978. pp. 526-531.

(Su 1980)

S.Y.H. Su; E. Ducasse.

A hardware redundancy reconfiguration scheme for tolerating multiple module failures.

IEEE Trans. on Computers. C-29, 3. Marzo 1980. pp. 254-258.

(Sundberg 1978)

C.W. Sundberg.

Erasure and error decoding for semiconductor memories.

IEEE Trans. on Computers. C-27, 8. Agosto 1978. pp. 696-705.

(Sundberg 1979)

C.W. Sundberg.

Properties of transparent shortened codes for memories with stuck-at faults.

IEEE Trans. on Computers. C-28, 9. Septiembre 1979. pp 686-690.

(Swanson 1980)

R. Swanson.

Matrix technique leads to direct error code implementation.

Computer Design. Agosto 1980. pp. 101-108.

(Szygenda 1971)

S.A. Szygenda; M.J. Flynn.

Coding techniques for faillure recovery in a distributive modular memory organization.

AFIPS Proc. SJCC. 1971. pp. 450-466

(Szygenda 1971)

S.A. Szygenda; M.J. Flynn.

Faillure analysis of memory organization for utilization in a self-repair memory system.

IEEE Trans. on Reliability. R-20, 2. Mayo, 1971. pp. 64-70

(Szygenda 1973)

S.A. Szygenda; M.J. Flynn.

Self-diagnosis and self-repair in memory: An integrated system approach.

IEEE Trans. on Reliability. R-22, 1. Abril 1973. pp. 2-12.

(Thatte 1977)

S.M. Thatte; J.A. Abraham.

Testing of semiconductor random acces memories.

Int. Symp. on Fault Tolerant Computing. 1977. pp. 81-87.

(Walker 1979)

W.K.S. Walker; C.W. Sundberg.

A reliable spaceborne memory with a single error and erasure correction scheme.

IEEE Trans. on Computers. C-28, 7. Julio 1979. pp. 493-500.

(Wakerly 1974)

J.F. Wakerly; E.J. McCluskey.

Design of low cost general purpose self diagnosing computers.

IFIP-74. 1974. pp. 108-111.

(Wakerly 1978)

J.F. Wakerly.

Error detecting codes, self-checking circuits and applications.

North-Holland. 1978.